University of Glasgow | School of Computing Science

Inheritance Locks

W. David Fröhlingsdorf

School of Computing Science
Sir Alwyn Williams Building
University of Glasgow
G12 8QQ

Level 4 Project — March 24, 2017

Abstract

During the last decades, computer systems gradually turned from sequential systems into parallel systems. This development has been happening in a small (multi-core processors) as well as in a large scale (networks and computer clusters). The underlying concept of multiple working units and shared resources, however, is essentially the same. In such systems, it is often necessary that a working unit requires exclusive access to a resource, or that a sequence of operations has to be performed as if it was one atomic operation. To fulfil these requirements we need locking protocols and similar concepts.

For this reason, much research has been done over the last fifty years and many concepts have been explored, including widely used ones such as mutexes, semaphores and transactional memory. Unfortunately, many protocols suffer under the risk of the so called deadlock, starvation and/or livelock in which the system is captured in an unfavourable state. To the best of our knowledge, no perfect locking protocol has been explored and further research is needed to entirely eliminate this risk occurring in parallel systems.

We therefore propose a new locking protocol, the Inheritance Lock Protocol, which prevents deadlock and starvation. We show the relation to existing locking protocols and show the correctness of the protocol. We also test the performance of the protocol and compare it to well known protocols using benchmark tests, showing that it is comparative to transactional memory in terms of performance. We also illustrate the limitations of the proposed protocol and test the protocol on a number of common applications.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Contents

[1] In this and the following sections we return to the process and resource terminology as more generic aspects of inheritance locking are discussed

Chapter 1

Introduction

Since at least the 1970s it has become standard in computing to run processes concurrently on the same system. This practice has many advantages such as improved processor utilization, increased security and flexibility by an intermediate operating system. Before the introduction of multi-processing, central processor unit (CPU) time was wasted while waiting for an Input/Output (I/O) operation to finish. Moreover, each program had to be written in such a way that it was tailored for the specific machine's hardware. This was not practical and the introduction of multi-processing together with operating systems was a milestone in computing.

For the last twenty years it has become common practice to fit computers with several CPUs or CPU-cores. For at least the last 10 years all mainstream computers have been sold with two or more cores. This means, processes can now be executed in parallel (that is, literally at the same time). Many programs make use of this and deploy several threads which individually execute separate parts of the program in parallel. This can increase the speed of the program and the system significantly. A similar development can be observed in the inter-connection of computers. Global computer clusters and data centres allow high connectivity and computational power. With the increase of cloud computing and the rapid growth of the internet (Figure 1.1), this development is gaining importance.

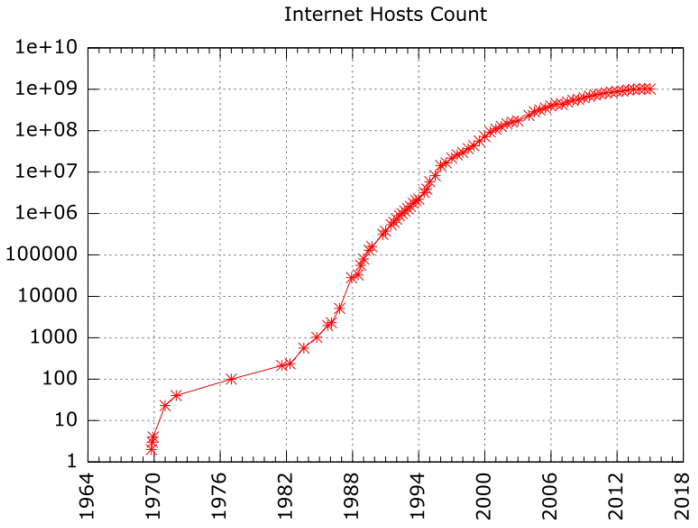


Figure 1.1: Internet hosts 1964-2018, **logarithmic scale**. Source: [25]

On the downside, there are many issues arising from these developments. This is probably best explained with an analogy. Consider a library; as long as there is only one visitor in the library he can read any book and

take as many books out as he likes. However, as soon as more visitors arrive, it becomes more likely that a visitor wants a book that has already been taken out by a different visitor. Similar to the analogy with the library; there are limited resources in computing systems and running several processes will lead to some clash sooner or later. For example, two processes can not read from a hard disk drive (HDD) at the same time as a standard HDD has only a single read/write head which can not physically be at two locations at the same time. It becomes even more complicated when considering reading and writing operations of processes. A thread might write a variable to memory which will immediately be overwritten by another thread which did a lengthy operation on the same variable. It is easy to see that these kind of scenarios can happen and lead to problems.

Locking protocols have been introduced to address this issue. The idea behind a locking protocol is that a process or thread can autonomously lock a required resource for as long as needed to securely finish the computation on this resource. The term, mutual exclusive access, is usually used to describe this kind of locking as no other process or thread has access to this resource for the given time.

Although mutual exclusion is very effective and is often used nowadays, it has limitations specifically when the locking protocol allows processes to freely lock and unlock resources. That is, a process can place locking and unlocking requests in any arbitrary order. This can lead to a state where two processes or threads wait indefinitely for each other to unlock a resource without making any progress. This system state is known as deadlock. We will discuss limitations of locking protocols in more depth in the next chapter.

1.1 Motivation

Using mutual exclusion for locking resources contains a deadlock risk which is hard to manage. Unfortunately, the alternatives to mutual exclusion are very limited and developers are therefore often forced to use mutual exclusive locks for concurrency control. This situation is not ideal and new locking protocols and libraries are therefore desirable.

Even though the deadlock problem is well known, it is hard to verify freedom of deadlock in a system. Hartonas-Garmhausen et al. showed that even safety-critical systems can suffer deadlock situations [17]. The fact that many major companies, like Oracle, Veritas or IBM [22, 38, 51], spot deadlocks in their software shows how important this problem is.

Hence, the motivation and aim of this project is to develop a new locking protocol which prevents deadlock, livelock and starvation as well as enhancing the opportunities of developers by providing the developed locking protocol as a C library.

1.2 Contributions

This report makes several contributions to the field:

- Overview of the state of art, discussing various approaches and key papers from the last 50 years
- Introduction of Inheritance Locks, a novel locking protocol which prevents deadlock, livelock and starvation as well as verification of the protocol using the SPIN model checker
- Performance benchmark tests for locking protocols

1.3 Outline

- Chapter 2 contains an overview of background information, previous and related work as well as an overview of the topic's terminology.
- Chapter 3 introduces the idea of the Inheritance Lock-Protocol, including a formal definition of the protocol.
- Chapter 4 shows how we verified the absence of deadlocks and starvation in the inheritance lock protocol using the SPIN model checker.
- Chapter 5 introduces the inheritance lock library, implemented in C, including implementation details.
- Chapter 6 contains a description of two benchmarks written to evaluate the performance of a locking protocol.
- Chapter 7 presents results of using the inheritance lock protocol in compiled software as a shared library.
- Chapter 8 discusses scenarios in which the inheritance lock protocol is useful as well as limitations of the protocol.
- Chapter 9 contains a discussion about the inheritance lock protocol and state of art of resource locking as well as a personal reflection of the project as a whole.

Chapter 2

Background & Related Work

The issue of resource locking was investigated extensively in the 1960s and early locking protocols were proposed and implemented. As outlined in the previous chapter, the deadlock problem is of huge importance and many protocols have been developed or amended to address this issue. In this chapter we discuss the various suggestions regarding resource locking. For this, we define in the following section the terminology which we use when talking about locking protocols. The terminology is widely used and should be familiar.

2.1 Terminology

- **Process**: Depending on the context a thread or process which can read/write from/to a resource.
- **Resource**: Usually a physical device or a memory space holding a variable. A resource is usually guarded by some kind of lock which means that a process can lock and unlock a resource. The process is granted exclusive access to the resource while it is holding the lock. Since the actual nature of the resource is not of interest, the term **lock** might be used interchangeably with resource.
- **Resource instance**: If the resource exists more than once and it is irrelevant for the requesting process which one of the equivalent resources will be assigned to the process, then we talk about a resource with a number of instances.
- **Lock-Request**: A process is trying to lock a resource. Whether this request is successful depends on the locking protocol and the circumstances.
- **Lock/Acquire**: A process was successful in its locking request.
- **Unlock/Release**: A process gives up a resource which can then be acquired by a different process.
- **Pre-emption**: A process is forced to temporarily give up an acquired resource. While a process is pre-empted, it is also stopped from running (i.e. it is blocked).
- **Availability**: A resource is available if no process has acquired it.
- **Deadlock**: Two or more processes, each holding at least one resource, waiting indefinitely on each other to release the resource.
- **Livelock**: The same as deadlock, but instead of being blocked the processes do some housekeeping work while indefinitely waiting for the other process to give up the resource. In other words, the process are engaged in a *busy-wait*.

- **Starvation**: A process is waiting for an unacceptable long time for an event to occur. In particular, the event is not guaranteed to eventually occur. Starvation is a common problem in locking protocols which prevent or avoid deadlocks.
- **Deadlock Prevention**: Deadlocks are not possible by the definition of the locking protocol.
- **Deadlock Avoidance**: Processes are scheduled in such a way that a deadlock can not occur. This usually requires some pre-knowledge about the required resources of the individual processes.

2.2 Related Work

This section contains an overview of related work in loose chronological order.

2.2.1 Havender's pioneer work

One of the earliest, highly cited and influential research works regarding locking protocols was presented by Havender [18]. He points out that there is a threat of deadlock in concurrent operating systems (that is, multi-tasking systems) and discusses the background of various locking mechanisms such as read/write locks in databases. Havender suggests four different locking protocols to prevent deadlocks from occurring.

- **Approach 1**: Ensure that each process locks resources in the same order. To ensure this, each resource has a unique priority assigned to it. Let m be the highest priority of all resource currently held by a process. The process is only allowed to lock a new resource if its priority is greater than m. Thus it is ensured that resources are locked in the same order and that deadlocks are prevented from occurring (proof omitted). This approach is usually known as hierarchical locking and a number of variations of it have been proposed for example by Dijkstra[8] and Muhanna[35].
- **Approach 2**: During a process's locking phase, require all needed resources at once and release all resource at once when the operations have been performed. This approach was adopted by many transactional approaches which were later introduced.
- **Approach 3**: If a locking-request can not be satisfied right away (because the requested resource is locked by another process), all resource held by the requesting process are pre-empted (that is, made available to be locked by any other process) and the requesting process can only resume once the requested and the previously locked resources are available.
- **Approach 4**: A process should be prepared to take an alternative route if it is not able to acquire a resource. Although, deadlocks are prevented from occurring when this approach is used, it is usually not feasible as alternative routes often end up in a loop and this approach is therefore likely to lead to a *livelock* situation.

2.2.2 Early work on deadlock avoidance

During the 1970s and 80s a distinction between the terms *deadlock prevention* and *deadlock avoidance* became conventional. Namely, deadlock prevention is a fundamental principle inside the locking protocol which makes deadlock impossible. Deadlock avoidance, on the other hand, is a mechanism which checks for each locking request individually whether it is safe in terms of deadlock to grant access to the resource. As long as it is impossible for a deadlock to occur, the system is in a so called safe state [53].

Early work from Habermann [16], Shoshani and Coffman [46] and Dijkstra's Banker's Algorithm [7] are important milestones for deadlock avoidance. We will not further discuss the individual approaches as they are not of relevance for this report.

While some authors, like Levine [27], argue that it is not always possible to distinguish between prevention and avoidance, protocols are usually still classified in these two categories today. Although the suggested deadlock avoidance protocols typically work well in theory, they usually lack in performance during locking and unlocking phases and often require more information about the running processes [12]. Prevention protocols on the other hand are usually more agile, however as Isloor and Marsland point out, they often block a process longer than necessary [23].

2.2.3 Dining Philosophers Problem

One of the most famous illustrations of the deadlock problem is the *Dining philosophers problem* which was introduced by Dijkstra as an example for his students [9, 8]. The problem states that five philosopher sit on a round table having dinner together. There is a fork between every two philosophers (see Figure 2.1). Each philosopher can either think or eat. A philosopher does not need anything for thinking, however, he needs his right and left fork in order to eat.

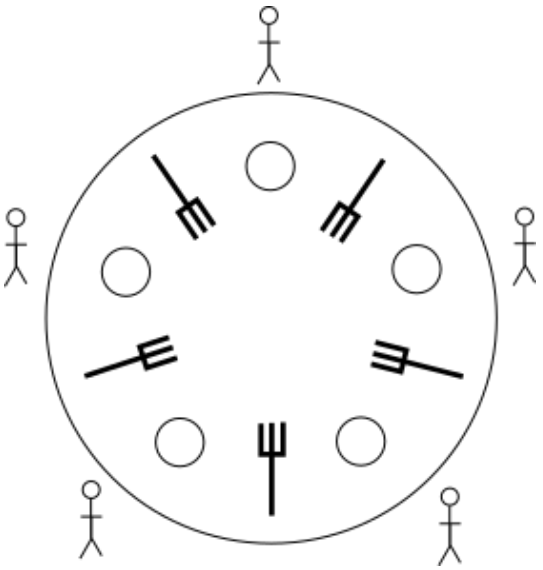


Figure 2.1: Dining Philosophers

Lets suppose that if a philosopher would like to eat he first picks up his right, then his left fork. He needs to wait if the according fork is already in use by his neighbour. Furthermore, a philosopher does not put a fork down until he has finished eating.

Consider now what happens when all five philosophers start eating at the same time. Each one is picking up their right fork and indefinitely waits for their left fork to become available. Therefore, the system is deadlocked.

Using the dining philosopher problem, it is easy to illustrate another issue that occurs in locking protocols, *Starvation*. Let us now consider an amended protocol so that the philosophers can not deadlock any more. For this, suppose that a philosopher is allowed to pick up his right and left fork if and only if (iff) both forks are not used by the corresponding neighbour. Notice that this is essentially the second approach suggested by Havender (Subsection 2.2.1). Now consider a philosopher who sits between two very hungry neighbours. His neighbours are almost constantly eating and have only very short thinking breaks. The philosopher can only start eating if

his two very hungry neighbours have one of their very rare thinking breaks **at the same time**. Because this event hardly ever occurs our philosopher has to starve and can not make any progress within a reasonable amount of time.

The dining philosophers problem is an excellent illustration as it can be directly mapped to computer systems. In particular, each philosopher is a process and each fork is a resource which needs to be locked in order to proceed into the *critical section*, i.e. the eating phase. It is therefore a suitable example for checking a given locking protocol for deadlocks and starvation.

2.2.4 Deadlock properties

The work by Coffman et al. [5] is considered to be the first which specifically classifies the conditions which are required for a deadlock to occur. To prove the correctness of a deadlock prevention protocol, it is sufficient to prove that one of those conditions can not hold. Coffman et al. show that the following four conditions are necessary for a deadlock:

- A process claims **mutual exclusion** over a resource.
- A process **waits for** a resource to become available while **holding** resource(s) which have already been assigned to it.
- Resources can not forcefully be removed, that is **pre-empted**, from a holding process.
- A **circular wait** of processes exists such that each process holds a resources which is requested by the next link in the chain.

2.2.5 Wait-For Graph

As already indicated by Coffman et al. [5], the relation between resources and processes can be modelled as a directed graph. The first fundamental work on this *wait-for graph* was done by Holt in 1972 [20]. Holt discusses system states in a graph theoretical sense. This means, a system is always in a unique state (Labelled: $S, T, U, V, W, ...$). By requiring or releasing a resource a process can move the system into a different state. For example in Figure: 2.2 process 1 can move the system from state S to state T or U. On the other hand, process 2 can not move the system into a different state from state S, therefore process 2 is blocked in S. We can see that in the given example process 2 is blocked in states S, U and V. Even more, should the system be in State U or V process 2 will be blocked forever as it is impossible to reach T from those states. We therefore say that process 2 is deadlocked in U and V. States which contain a deadlocked process are called deadlock states. States in which all processes are deadlocked, in the example State V, are called total deadlock state.

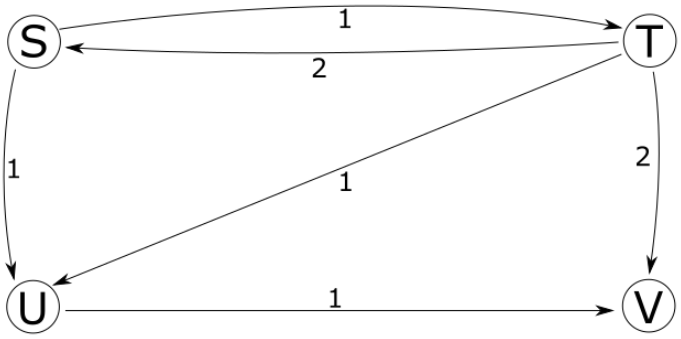
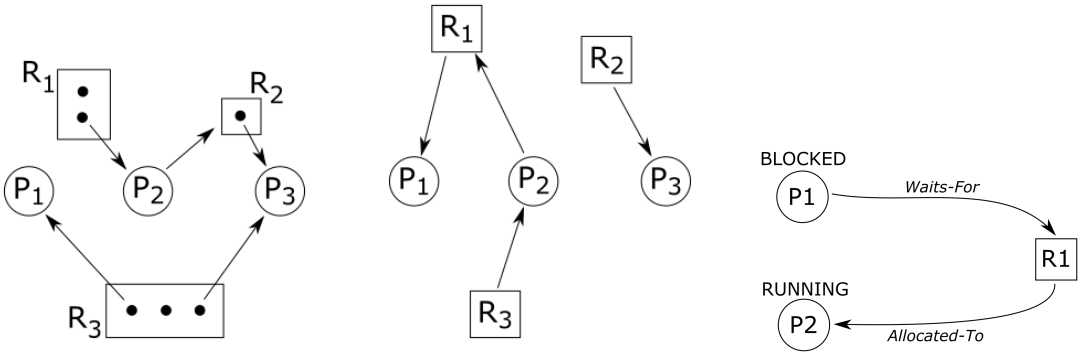


Figure 2.2: System State Transition Graph

It is fairly easy to see how this graph is of importance for deadlock avoidance as those protocols are actively trying to avoid transitions into deadlock states and therefore keep the system deadlock free (i.e. in safe states). Unfortunately, it is necessary to obtain more information from each process in order to foresee transitions between system states. Moreover, depending on the complexity of the system, the state space can easily grow exponentially. It is therefore hard to justify this approach for general purpose libraries.

Holt moves on to analyse specific system states using wait-for graphs. While he distinguishes in his work between reusable and consumable resources, more recent work usually does not consider consumable resources. We will therefore only consider an established version of the wait-for graph in which all resources are reusable. Figure 2.3a shows such a version of a wait-for graph. The circular nodes, named P_i, are representations of processes. The rectangular nodes, named R_i, are resources. Each resource might have more than one instance. The number of instances per resource are visualised with black dots inside the resource node. In practice a resource could be a physical drive or swap space. However, usually a process is not concerned with which drive or swap space would be allocated to it. Therefore they can be bundled as one resource and each physical device is an instance of that resource. An edge from a resource instance to a process, means that this instance has been allocated to the process (for example R_2 to P_3). An edge from a process to a resource means that the process is requesting an instance of the resource (for example P_2 to R_2). If no further instance of a resource is available, a requesting process is blocked and has to wait for an instance to be released by a process (Figure 2.3c). Depending on the used locking protocol, a process might request more than one resource at once.



(a) Established Wait-For Graph

(b) Simplified Wait-For Graph

(c) Resource - Process relation in Wait-For graph

Figure 2.3: Wait-For Graphs

The situation nowadays has significantly changed because locking protocols are now usually used for synchronizing multi-threaded programs (with the exception of operating systems). This means, a resource is a specific piece of data on which a number of sequential operations should be performed as a single atomic operation by a thread. Hence it does not make sense to argue that resources in such a scenario have more than one instance. Many wait-for graphs are therefore simplified (see Figure 2.3b) and assume the principle: One resource, one instance, one lock.

Wait-for graphs in other approaches

As Minoura [34] points out even older locking protocols, like Havender's scheme [18] and Habermann's algorithm [16], can easily be modelled and refined using wait-for graphs. Locking protocols using wait-for graphs have been proposed by Datta et al. [6], Naimi et al. [36], Barbosa [3] and Oliveira and Barbosa [37] to name just a few.

A further deadlock prevention approach uses Petri Nets which are similar to wait-for graphs as they are based on directed bipartite graphs. Significant deadlock prevention work using petri nets has been done by Ezpeleta et al. [11] and by Li and Zhou [28].

2.2.6 Deadlock detection

Another common approach to the deadlock problem is *deadlock detection*. This means, deadlocks can occur but the system or the system administrator will resolve the issue by hand, usually by killing one or more processes which are involved in the deadlock. Nowadays, most ordinary operating systems use this approach because inter-process deadlock is uncommon and running a deadlock prevention or avoidance algorithm would result in too much performance overhead [47].

For example, Microsoft has introduced in their operating systems, Windows, the capability to analyse the wait chain of a process which can be done through the Task Manager since version Windows 7 [32]. While this is a retrospective approach, there are also deadlock detection algorithms which can be run preventively. Williams et al. [52] developed a static deadlock detection checker for Java libraries which can directly be run on Java source code. Their findings suggest that their checker delivers usable results and is often able to verify deadlock freedom in libraries.

2.2.7 Locking in databases

A major focus in deadlock prevention/avoidance is set on databases as they have to provide as much parallelism as possible while guaranteeing data consistency. Several key papers discussing locking in databases were published around 1980. Rypka and Lucido [43] discuss the difference between read and read/write locks in logical resources. The differentiation between these two types of locks plays an important role in database concurrency control.

Most operations in databases are a sequence of read/write accesses which are performed as a single atomic *transaction*. It is usually desired that all accessed data stays consistent (that is, unchanged by other transactions) for the duration of the transaction. Menasce and Muntz study the possibilities to detect a deadlock in such a transactional environment [31]. Similarly, Silberschatz and Kedam explore ways to ensure deadlock freedom in a transactional environment. Their work has a particular focus on graph based solutions in two-phase locking databases [48]. Two-phase locking is an approach in which all required locks are requested collectively once a transaction has been committed. All read and write operations are only performed once the collective request has been successful. More recent work on transactional deadlock prevention has been performed by Lou et al. Their approach uses timestamps and the results show performance improvements, particularly in distributed systems, compared to previous approaches [30].

Transactional concurrency control has been successfully implemented on applications other than databases too. In recent years much research on *transactional memory* has been performed. The idea behind transactional memory is to perform a block of code in a single atomic step similar to a transaction in a database. Therefore, transactional memory guarantees data consistency while ensuring freedom of deadlock. However, no side effects, such as I/O operations, are allowed inside a transactional block and starvation is possible as pointed out by Herlihy and Moss [19]. Peyton Jones discusses transactional memory in Haskell [40]. The latest versions of the GNU Compiler Collection (GCC) have a build-in support of transactional memory as well [13, 44].

2.3 State of the art

As discussed, early research in the field was mostly concerned with physical resources and competing process in an operating system. However, this issue has turned out not to be significant enough for most operating systems and inter-process deadlock is therefore mostly ignored. This is particularly true as the capacity of physical resources has increased and the operating system takes care of fair scheduling for these resources.

The issue of data consistency has been well understood and addressed in databases. By using transactions and two-phase locking, freedom of deadlock is ensured too. On the other hand, there is great interest in exploring further approaches allowing a performance increase.

The recent increase of multi-threading processes has lead to a high demand for locking protocol libraries. While classic libraries, like the POSIX-Thread (Pthread) or the C++ library, contain locks to ensure data consistency, they do not provide mechanisms to allow deadlock prevention, avoidance or detection. Transactional memory can solve the issue of deadlock prevention, however, it has own limitations. Some of the most important limitations of transactional memory are, not allowing side-effects and the risk of starvation.

The trend of improving performance by multi-threading an application will certainly continue. A new locking protocol and library for intra-process locking (that is locking within the same process memory space) is therefore desirable. For this reason, we will introduce Inheritance Locks, a novel protocol which prevents deadlocks, livelocks and starvation in the next chapter.

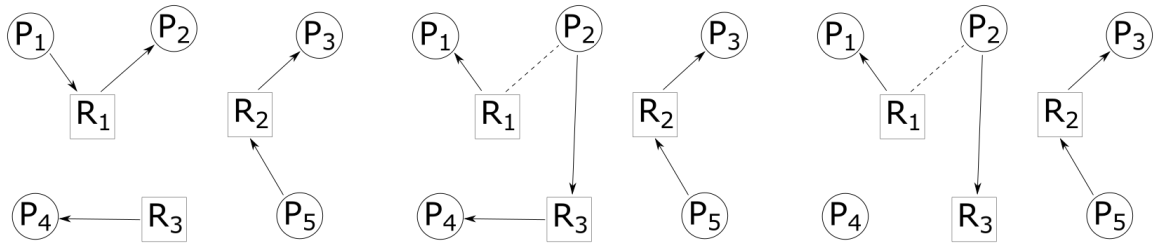
Chapter 3

Inheritance Locks

In the previous chapter we discussed related work and background information of locking protocols. We pointed out that many locking protocols suffer the risk of deadlock or lead to starvation. In this chapter we will introduce Inheritance Locks, a novel locking protocol which prevents deadlocks and starvation. We will provide examples to illustrate the semantics of the protocol.

3.1 Origin

As discussed in Section 2.2.1, Havender [18] introduces a number of approaches to address the deadlock problem. We will particularly focus on **Approach 3** and show that it can lead to *starvation*. The approach states that if a locking-request can not be satisfied right away (because the requested resource is locked by another process), all resources held by the requesting process are pre-empted (that is, made available to be locked by any other process) and the requesting process can only resume once the requested and the previously locked resources are available. Let us illustrate this approach with an example using a wait-for graph.



(a) R_1 is assigned to P_2

(b) P_2 requests R_3. Therefore R_1 is pre-empted and assigned to P_1

(c) Even after P_4 releases R_3, P_2 can not continue until R_1 is available again.

Figure 3.1: Successive points in time (a),(b),(c) - Starvation in Havender's Approach 3

Figure 3.1a shows a system with five processes and three resources. Resource R_1 is assigned to process P_2, R_2 is assigned to P_3 and R_3 to P_4. The processes P_1 and P_5 are currently blocked, because they are waiting for R_1 and R_2 respectively to become available. Let us now assume P_2 reaches a critical section in which it is required to additionally request R_3 and later R_2. Since R_3 is currently assigned to P_4, P_2 has to wait and becomes blocked. Therefore R_1 gets pre-empted and assigned to P_1. Figure 3.1b shows this situation. Next, P_4 releases R_3, however, P_2 can still not acquire R_3 because the pre-empted resource R_1 is still used by P_1 as illustrated in Figure 3.1c. It is easy to see that a similar situation will occur once P_2 has successfully (re-)acquired

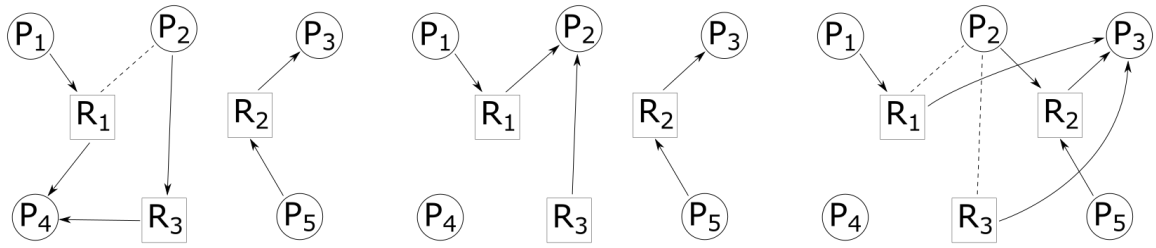
R_1 and R_3 and tries to lock R_2. P_2 can only enter the critical section iff R_1, R_2 and R_3 are available at the same time. However, since five processes are competing for the resources, there is no guarantee that this situation will ever occur. The waiting time might become unacceptable, even if the situation eventually occurs. Hence, we can conclude that in Havender's Approach 3 starvation is possible.

3.2 Approach

As shown in the previous section, Havender's Approach 3 can lead to starvation. We propose therefore Inheritance Locking, an amendment of the approach which is free from *starvation*.

Definition 3.2.1. Informally: If a locking-request can not be satisfied right away, all resources held by the requesting process are pre-empted and assigned (inherited) to the process which is holding the requested resources. Once the requested resource is unlocked all inherited resources are pre-empted and reassigned to the requesting process.

Let us use the previous example, shown in Figure 3.1, to illustrate the improved behaviour of inheritance locking. Let us assume that the initial situation, shown in Figure 3.1a, is the same.



(a) R_1 is inherited by P_4.

(b) Once P_4 releases R_3, P_2 holds both, R_1 and R_3.

(c) Because P_2 requests R_2, which is locked by P_3, R_1 and R_2 are inherited by P_3.

Figure 3.2: No starvation in Inheritance Locks

Other than in Havender's Approach 3, P_1 will not be able to lock R_1 once P_2 becomes blocked, because R_1 is inherited by P_4 (Figure 3.2a). Once P_4 releases R_3, R_1 will be reassigned to P_2 and P_2 will also be successful in locking R_3 (Figure 3.2b). P_2 can then move on to request R_2, currently held by P_3, which means R_1 and R_3 will be inherited by P_3 (Figure 3.2c).

3.3 Illustrative example

Since the definitions of the previous sections are fairly abstract we give an informal illustrative example in this section in the hope that this clarifies how Havender's Approach 3 differs from Inheritance Locks. Table 3.1 and Figure 3.3 are provided to make the relation between the example and the locking protocols clearer.

Consider three friends, Alice, Bob and Charlie, who share a textbook, a pencil and a highlighter pen. In this analogy Alice, Bob and Charlie are processes and the textbook, pencil and highlighter pen are resources which can not be used by more than one person (process) at a time. Let's consider the following sequence of events: (1) Alice is reading the textbook (resource textbook allocated to process Alice), (2) Bob picks up the highlighter pen and the pencil (process Bob locks resources highlighter pen and pencil) and (3) wishes to highlight passages in

Table 3.1: Illustrative example as table

(a) Havender's Approach 3

	Time				
	(1)	(2)	(3)	(4)	(5)
Textbook	A	A	A	A	-
Highlighter	-	B	-	C	C
Pencil	-	B	-	C	C

(b) Inheritance Locks

	Time				
	(1)	(2)	(3)	(4)	(5)
Textbook	A	A	A	A	A
Highlighter	-	B	B	B	A
Pencil	-	B	B	B	B

the textbook (process Bob requests resource textbook) and take some private notes, (4) Charlie would like both pens (process Charlie request resources highlighter pen and pencil) to go over his private notes, (5) finally Alice wants the highlighter pen (process Alice request resource highlighter pen) to highlight a sentence that she just read in the textbook.

In Havender's Approach 3, Bob puts away the pens in (3)(resources highlighter pen and pencil are pre-empted from process Bob, because the requested resource textbook is not available) and would only try to get them back once the textbook becomes available. This means Charlie picks up the pens (process Charlie acquires resources highlighter pen and pencil) and works on his private notes in (4). Alice and Bob (process Alice is blocked while waiting for resource highlighter pen, process Bob is blocked while waiting for resources highlighter pen and pencil) have to wait for Charlie in (5) to finish his work before Alice could pick the textbook up again and highlight her sentence.

In particular Bob suffers from starvation in this approach because he wants all resources and can only continue iff all of them are available at the same time. It also feels counter-intuitive why Bob puts the pens away (3) if the textbook is not immediately available. Let us now observe what happens in this scenario if the Inheritance Locking Protocol is used instead.

Bob does not put down the pens in (3) and Charlie has to wait (process Charlie becomes blocked while waiting for resources highlighter pen and pencil to become available) for Bob in (4). In (5) Bob lends the highlighter pen to Alice (resource highlighter pen inherited by process Alice) as she is having the textbook that Bob is waiting for. In fact, Bob is smart enough to see that if he would not lend Alice the highlighter pen, Alice would not give up the textbook meaning they are in a deadlocked situation. Intuitively, Inheritance Locking enforces reasonable and fair resource handling while preventing deadlocks from occurring.

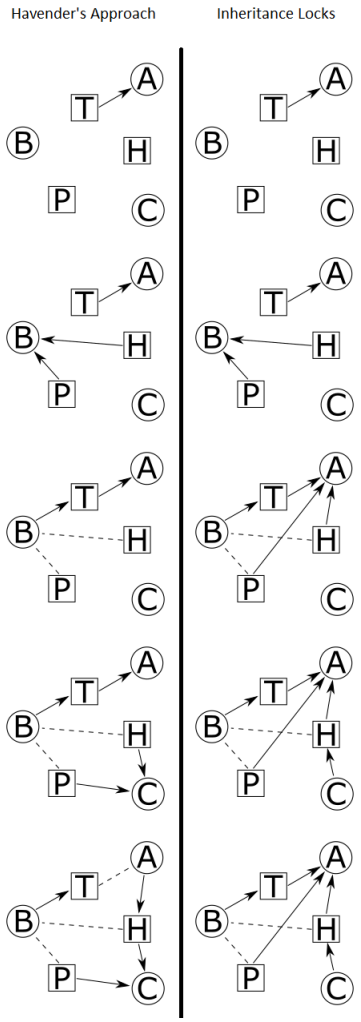
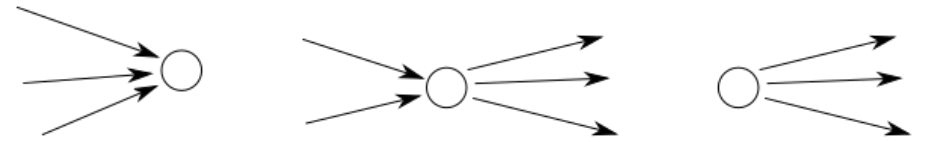


Figure 3.3: Wait-For graph of illustrative example in chronological order.

3.4 Definition using Graph Theory

In this section we will refine and improve the definition of inheritance locking (Definition 3.2.1) using graph theory. Directed graphs can be used to describe *wait-for graphs*. We will therefore

use directed graphs to define inheritance locks. These results will give a clear formal definition which will be used throughout this report. For completeness, we will first give a definition of graph theory terminology, in particular of directed graphs.



(a) Sink (outdegree = 0). (b) Internal node (indegree = 2, outdegree = 3) (c) Source (indegree = 0).

Figure 3.4: In/Out-Degree examples in a directed graph

3.4.1 Directed Graphs

A *directed graph* $G = (N, E)$ consists of a set of *nodes* (sometimes called vertices) N and a set of *directed edges* E such that each edge goes from one particular node to another $E \subseteq N \times N$ (self-referencing is possible). The *outdegree*, denoted by $deg^+(n)$, of a node n is the sum of outgoing edges from n. Formally

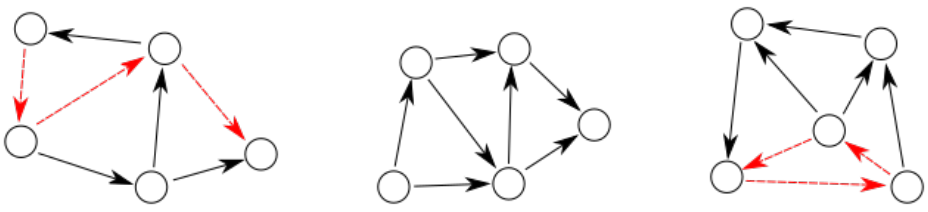
$$deg^+(n) = \sum_{v=n_1}^{N} (e|e \in E \wedge e = (n, v))$$

The *indegree*, denoted by $deg^-(n)$, of a node n is the sum of ingoing edges to n. Formally

$$deg^-(n) = \sum_{v=n_1}^{N} (e|e \in E \wedge e = (v, n))$$

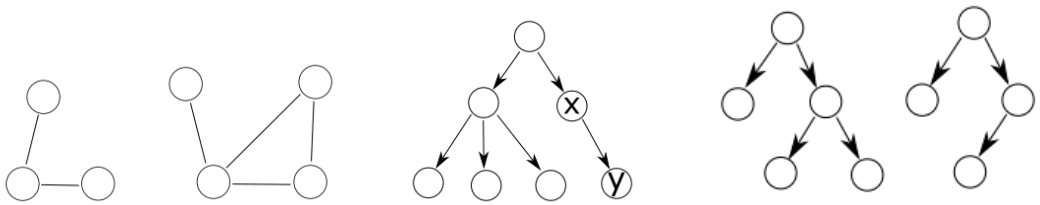
A node n with no outgoing edges $deg^+(n) = 0$ is called *sink*. A node n with no ingoing edges $deg^-(n) = 0$ is called *source*. A node that is neither a sink nor a source is called an *internal node*.

A *path* is a sequence of edges such that each subsequent edge starts at the node where the previous edge ended. An *directed acyclic graph* (DAG) is a graph in which there is no path such that it visits a node twice.



(a) Path in a directed graph (b) Acyclic directed graph (c) Cyclic directed graph

Figure 3.5: Paths and cycles in a directed graph



(a) Graph with two components (b) Tree (c) Forest

Figure 3.6: Components and trees in graph theory

A graph consists of two or more *components* if there are two nodes which can not be connected by a path. A directed acyclic graph is called *tree* if each node has an indegree of exactly one, except for one node which is a source and is called the *root* node. All edges in a tree show a parent-child relation such that the outgoing node is the parent of the ingoing node. In Figure 3.6b node x is the parent of node y. A directed acyclic graph with several trees is called a *forest*. In a forest one to all nodes can be a root node (source). Each tree is a separate graph component.

3.4.2 Relation to Inheritance Locks

Using the definitions from the previous subsection we will now define inheritance locks as a forest of transpose trees in the *wait-for graph*. We will first define the term transpose tree and then the properties of inheritance locks using the former.

Definition 3.4.1. A *transpose tree* is a tree with each edge reversed (sometimes called converse graph). In particular each edge goes from a child node to a parent node. A *root* node is a sink, every other node n has an outdegree of 1: $deg^+(n) = 1$. All other basic properties of a tree are preserved. Figure 3.7 shows an example of a transpose tree.

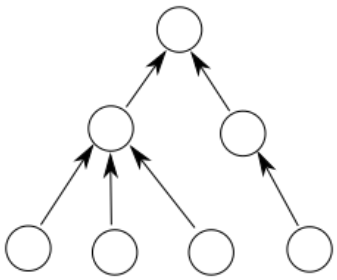
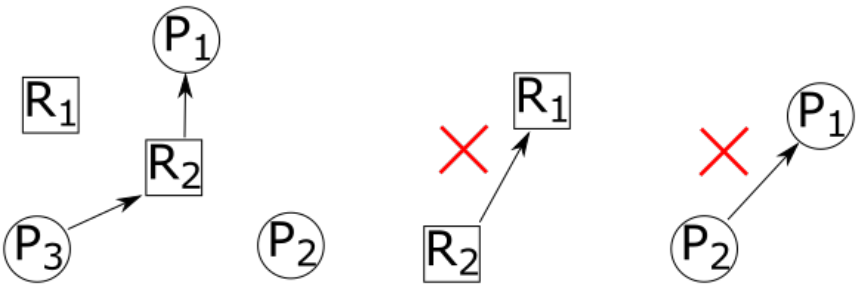


Figure 3.7: Transpose Tree

Note: For the rest of this report we will use the term tree if we mean a transpose tree, unless specifically stated otherwise.



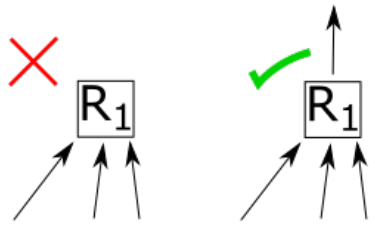
(a) R_1 is available, R_2 is assigned to P_1, P_3 is blocked while waiting for R_2, P_1 and P_2 are running.

(b) Not allowed, edge from resource to resource.

(c) Not allowed, edge from process to process.

Figure 3.8: Inheritance Locks - Graph examples

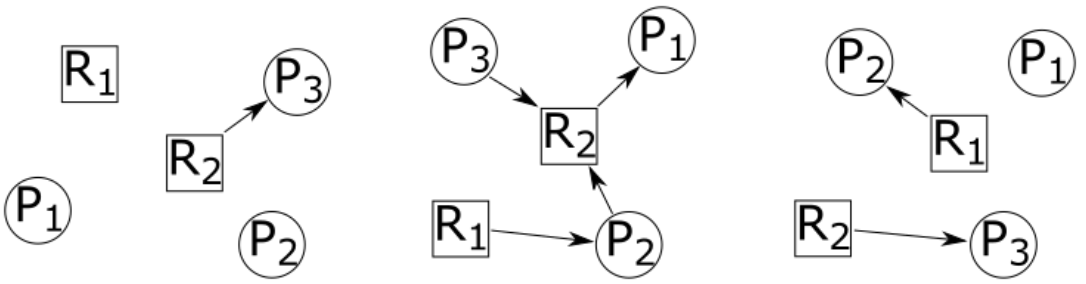
Let the graph $G = (N, E)$ consist of a set of processes P and a set of resources R. Let the nodes of the graph be $N = P \cup R$ and the directed edges be $E \subset ((P \times R) \cup (R \times P))$ (Figure 3.8). An edge $e = (p, r)$ means that process p is waiting for resource r to become available (wait-for relation). An edge $e = (r, p)$ means that resource r is assigned to process p (assigned-to relation). Each node in N can have an outdegree of 0 or 1 ($deg^+(n) = 0 \vee 1, n \in N$). A process with an outgoing edge is blocked while waiting for a resource. A process with no outgoing edge is running. A resource with an outgoing edge is assigned to a process. A resource with no outgoing edge is available. No process can wait for a resource if the resource is available $deg^+(r) = 0 \Rightarrow deg^-(r) = 0 \wedge deg^-(r) > 0 \Rightarrow deg^+(r) = 1, r \in R$; in other words if a resource node has children, it has to have a parent (Figure 3.9).



(a) (b)

Figure 3.9: A resource node must have a parent if it has children.

Let us now consider the locking request of inheritance locks. There are three possible scenarios if a process p wishes to lock a resource r: If r is available ($deg^+(r) = 0$), r can be directly assigned to p with an edge $e = (r, p)$ (Figure 3.10a). If r is not available ($deg^+(r) = 1$), the root node of r can either be p or different to p. If $root(r) = p$, r has been assigned to p (directly or indirectly through inheritance) and p can continue (Figure 3.10b). If $root(r) \neq p$ (Figure 3.10c), p becomes blocked and has to wait for r to become available($e = (p, r)$), meaning p is added to the tree of $root(r)$.



(a) R_1 is available.

(b) P_1 has inherited R_1.

(c) P_1 will have to wait for R_1 to become available.

Figure 3.10: P_1 wishes to lock R_1, three scenarios are possible.

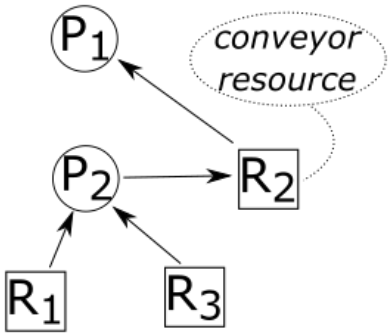


Figure 3.11: Conveyor Resource - R_2 makes R_1 and R_3 available for P_1

The release phase of inheritance locks is of specific concern, because a resource r should only be unlocked iff all resource children of r are unlocked too. That is, if inherited resources are still in use, unlocking a resource which made the inherited resource available (we call this the *conveyor resource*, Figure 3.11) should not be possible. Note that locking and unlocking inherited resources does **not** have an effect on the wait-for graph.

An easy solution for this issue is to enforce that locking and unlocking is done in a stack fashion (Last-In-First-Out). For example, if a process P_1 locks, in this order, resources R_2, R_1 and R_3, it has first to unlock R_3 then R_1 and eventually R_2. Because inherited resources can only be locked after the conveyor resource has been locked, locking in this fashion ensures that all inherited resources are unlocked before the respective conveyor resource is unlocked (Figure 3.12). Alternative of approaches of unlocking are possible which we will discuss in Chapter 8.

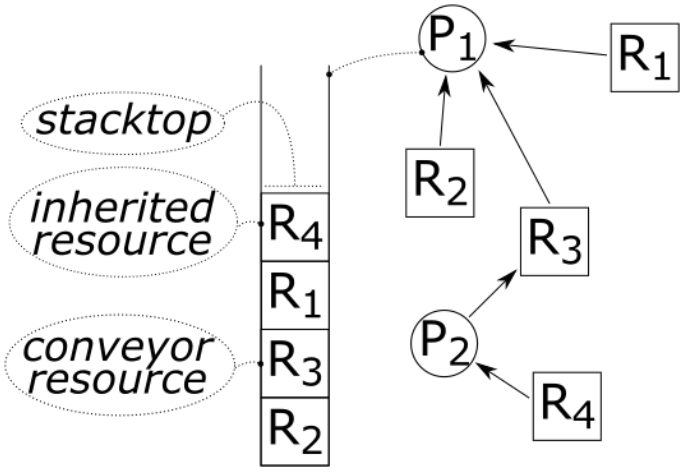


Figure 3.12: Resource stack of process P_1

Since each node can have an outdegree of 0 or 1 and at each locking request it is checked whether the root of the resource's component corresponds to the requesting process, it is a fairly obvious that this kind of locking leads to a forest in the graph. Cycles are therefore not possible. Using the work by Coffman et al. [5], introduced in Subsection 2.2.4, we can conclude that deadlock is prevented since pre-emption is partially allowed and, more importantly, circular wait is not possible. We will omit a formal proof of deadlock and starvation prevention.

In this chapter we have introduced a novel locking protocol, Inheritance Locking, which prevents deadlock and starvation. To the best of our knowledge, this protocol has not been proposed before. In the next chapter we will verify the correctness of inheritance locks using the SPIN model checker.

Chapter 4

SPIN Model Checking

In the previous chapter we introduced a novel locking protocol, Inheritance Locking, which we claim prevents deadlock and starvation. In this chapter we will initially verify the correctness of this protocol using the *SPIN* (Simple Promela Interpreter) model checker. SPIN is a popular tool for detecting software defects in concurrent system designs using properties expressed as *Linear temporal logic* (LTL) [21]. LTL allows properties which are time constraint such as, eventually s holds, t holds until u holds etc. This makes SPIN very suitable for testing a locking protocol. Models for SPIN are typically written in the *Promela* (Process Meta Language) verification modelling language.

4.1 Model design

SPIN creates a state space to check all possible routes of a concrete model, that is, for model x check that properties y and z hold. Unfortunately, it is not straightforward to verify properties which should hold in any arbitrary model. Therefore, it was not possible to efficiently create a model which would verify inheritance locks with any number of threads and any number of locks. Hence, we decided to run model checks against a range of threads and locks instead. In particular, for each specific number of threads and locks a separate model had to be created and verified by the SPIN model checker.

For the model creation we implemented a program in Haskell which generates the Promela model for a number of threads and locks given as arguments to the program. For the number of locks, a nested part of the model had to be created. The number of threads, on the other hand, just required a change of a global constant in the model.

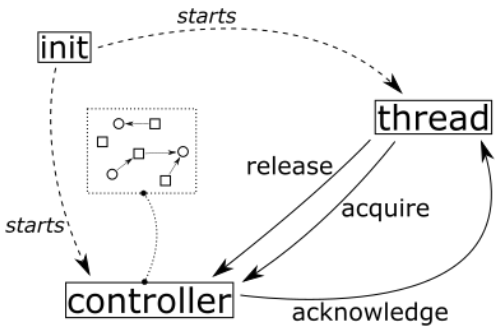


Figure 4.1: SPIN model

The model itself consists of three process types. An init process which initialises the variables and processes, a thread process which randomly locks and unlocks resources in a stack fashion and a controller process which manages a *wait-for graph* representation and handles locking and unlocking requests from thread processes. Locking and unlocking requests from a thread can be made by placing an according message on an acquire and release channel respectively. The controller process reads from the acquire and release channels, processes the request and if the request was successful answers with an acknowledge message to the according thread. Figure 4.1 shows the model as a diagram.

4.2 Properties

The model had to be verified for two properties, mutual exclusion and progress. We checked the mutual exclusive property by using a variable, named $critical$, which is incremented every time a thread locks resource number 1 and decremented once it is unlocked or the holding thread blocks. An LTL property was used to check that $critical$ can only be 0 or 1, meaning that no more than one thread is in the critical section.

We checked the progress property by placing a label on top of the thread loop called $label\_thread$. Using an LTL property we could define that the first thread should eventually always make it back to the label, meaning that there are no *deadlocks* or *starvation* in the system. It is sufficient to check this property with a single thread as all threads execute identical code. Therefore, if the property hold for one thread, it holds for all of them. We had to enable weak fairness in order for this property to be verified. Weak fairness means that a thread should eventually run if it is **constantly** ready to run. That is, a non-blocked thread should eventually get some processor time so that it can progress. We checked with models of the range 2 to 10 threads and 1 to 10 locks as shown in Table 4.1.

Figure 4.2 shows the actual LTL properties as described. p is true when thread $p1$ is at the label $label\_thread$. q is true iff the variable $critical$ is equal to 0 or 1. $property1$ says that it should **always** be the case that, p is **eventually** true and q is true.

```
#define p (thread[p1]@label_thread)
#define q (critical == 0 || critical == 1)

ltl property1 {[]((<>p) && q)}
```

Figure 4.2: LTL Properties

4.3 Setup

Because the runtime of the verification grows exponentially in relation to the number of locks and threads, running the verification on a single machine was not suitable. Therefore, we designed the verification to be run on several machines in the Computer Science Lab during the Christmas Holidays. In particular, we wrote a script `login.exp` (Appendix A.1) in *expect*, an extension to the *Tcl* scripting language [29], to log in to a machine via secure shell (SSH), create a directory for this machine, copy all relevant files into the directory and run a number of verification tests for a given number of locks. In particular, each machine would first verify the range [2-10] threads with the number of specified locks. Then, every machine would verify the range [2-10] threads with [1-3] locks. This work was done by the `remoteArchive.sh` shell script (Appendix A.2) using the "nohup &" command. The nohup command can be used to start execution of a program that should not be terminated if the shell gets closed, meaning that the expect script could logout of SSH once the execution was

started. The expect script itself would be started by a shell script, `runExpect.sh` (Appendix A.3), which takes as arguments a password, the prefix of the room and the number of machines in the room. This shell script ensured that each verification was run at least three times as a backup and consistency check. Taken together, this means 21 machines had to run verifications, in particular Table 4.1 shows how often each verification was planned to be run.

Table 4.1: Scheduled number of verifications

		Threads								
		2	3	4	5	6	7	8	9	10
Locks	1	21	21	21	21	21	21	21	21	21
	2	21	21	21	21	21	21	21	21	21
	3	21	21	21	21	21	21	21	21	21
	4	3	3	3	3	3	3	3	3	3
	5	3	3	3	3	3	3	3	3	3
	6	3	3	3	3	3	3	3	3	3
	7	3	3	3	3	3	3	3	3	3
	8	3	3	3	3	3	3	3	3	3
	9	3	3	3	3	3	3	3	3	3
	10	3	3	3	3	3	3	3	3	3

It was expected, that the verification of the range [1-3] locks would be executed relative quickly regardless of the number of threads. For simplicity every of the 21 machines was therefore scheduled to check this range after checking the range [2-10] threads with the uniquely specified number of locks. This meant, fewer machines were needed and the script could be kept simple for the cost of running more verifications than necessary on the range [1-3] locks.

The verifications were scheduled to be run using high compression as the amount of memory on the machines was a limiting factor. Using a higher compression rate meant having a trade-off between space and time. Table 4.2 shows the technical properties of the machines.

Table 4.2: Machine details

Memory	3.6 GiB
Processor	Intel Core i5-3470 CPU @ 3.20GHz x 4
OS	CentOS Linux 7 - 64-bit
Graphics	Intel Ivybirdge Desktop
GNOME	Version 3.14.2
Disk	115.8 GB

4.4 Results

We could verify a number of models using the described setup. Table 4.3 shows how often each (lock, thread) pair could be verified. Each table cell also indicates, in brackets, how many seconds on average the verification took. No verification failed. Note that Table 4.3 is directly related to Table 4.1.

Unfortunately, many verifications could not finish as machines' memories were exhausted and we had to abort execution on a number of machines which had not finished after approximately 10 days, because the lab machines had to be used by returning students. Given more time and machines with more memory space, we

claim that even more models could have been verified. However, much more memory space would be needed to verify the entire matrix.

Even though not every model could be verified, all models had been attempted to be verified at least once and all finished verifications were positive. These results therefore clearly suggest that inheritance locks prevent deadlock and starvation while preserving mutual exclusion.

Table 4.3: Number of verifications (average runtime in seconds)

Locks \ Threads	2	3	4	5	6	7	8	9	10
1	9 (0.003)	7 (0.08)	6 (1.542)	5 (26.68)	N/A	N/A	N/A	N/A	N/A
2	7 (0.06)	7 (3.186)	6 (140.5)	6 (151000)	N/A	N/A	N/A	N/A	N/A
3	7 (0.541)	6 (53.517)	5 (211800)	N/A	N/A	N/A	N/A	N/A	N/A
4	2 (4.275)	2 (869500)	N/A	N/A	N/A	N/A	N/A	N/A	N/A
5	2 (41.9)	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
6	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
7	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
8	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
9	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
10	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

In this chapter we presented a Promela model for the inheritance locking protocol. Using SPIN, we verified many instances of the model on a computer cluster with the results clearly supporting our hypothesis that Inheritance Locks prevent deadlock and starvation while preserving mutual exclusion. In the next chapter we will present an implementation of inheritance locks in the C programming language.

Chapter 5

Inheritance Lock Library

In the previous two chapters we introduced inheritance locks, a novel locking protocol. We claimed that the protocol is free of deadlock and starvation while preserving mutual exclusion and supported this hypothesis with the results of checking the protocol with the SPIN model checker.

In this chapter we present an implementation of inheritance locks in the C programming language. We will first introduce aims and justify design decisions in Sections 5.1 and 5.2 respectively. In Section 5.3 we introduce the Application Programming Interface (API) of the library and in Section 5.4 we present the library implementation using high level pseudo code.

5.1 Aims

The aim of implementing inheritance locks was mostly of experimental nature. We wanted to show that the inheritance locking protocol can be implemented and has advantages over other well established locking methods. We did not aim to optimise the library. However, we were careful to avoid highly complex algorithms to keep running times reasonable. We are aware of many optimisation possibilities of the protocol which we will discuss in Chapter 8.

5.2 Design decisions

The C programming language was chosen for the implementation of the library as it is a well established, popular and cross-platform language. C++ is fully downward compatible to C, so C source code can be directly used in C++ code. That means the library can be used by a wider audiance. The TIOBE index, which is often used to identify the popularity of a programming language, lists C and C++ constantly among the top five programming languages over the last decade [49].

C gives much freedom to experiment and directly implement ideas without complications. This is because C is a weakly typed, fairly low-level language and it does not require a virtual machine or similar capabilities to run. This property allows reliable and reproducible performance measurements too.

The implementation is based on the POSIX standard, in particular POSIX Threads (pthreads), as this standard is widely established and supported by most operating systems. It also means that benchmark tests could be run against pthread mutexes (Chapter 6 and 7).

The GNU Compiler Collection (GCC) was used for compilation for a number of reasons. Firstly, it is a widely used and well established compiler meaning that performance tests should be easily reproducible. Secondly, GCC has a native support for *transactional memory* which we used in benchmark tests as shown in Chapter 6. The `-O3` optimisation flag was used for compilation to improve performance of the library.

5.3 API

We aimed to keep the Application Programming Interface (API) as easy and straightforward as possible. The entire API of the library is bundled in a single header file, `inheritance_lock_api.h` (Appendix B.1), which declares the generic `INHERIT_LOCK` type as a void pointer as well as the following five functions:

- `INHERIT_LOCK create_lock(INHERIT_LOCK*);` Creates a new inheritance lock.
- `INHERIT_LOCK create_rec_lock(INHERIT_LOCK*);` Creates a new recursive inheritance lock (can be locked more than once by the same thread).
- `int acquire(INHERIT_LOCK);` Calling thread tries to acquire the inheritance lock.
- `int release(INHERIT_LOCK);` Calling thread tries to release the inheritance lock.
- `int destroy_lock(INHERIT_LOCK*);` Destroys an inheritance lock.

To use the library, the programmer has to include this header file and link to the `inheritance_lock.o` file and to pthreads with the `-lpthread` flag.

5.4 Implementation

In this section we will present the basic implementation of the library using pseudo code. We will only present the code for the acquire and release functions here as they correspond to the core logic of inheritance locking. In subsection 3.4.2 we discussed the relation between inheritance locks and directed acyclic graphs (DAG). The pseudo code largely reflects this relation. Moreover, it is important to notice, that no two threads should modify the *wait-for graph* at the same time as this could cause a race condition. The implementation therefore uses a global pthread mutex, called `graphLock`, to grant mutual exclusive access for the wait-for graph. The `graphLock` is guaranteed not to be prone to deadlock, because no further pthread mutexes are locked inside the *critical section*. As shown in Chapter 6 and discussed in Chapter 8 the `graphLock` is a bottleneck of the library, though. The stack of held locks for each thread is an array of fixed size and can hold up to 32 locks. If the thread acquires more locks than the array can hold, a dynamic stack, implemented as a doubly linked list, is used for all locks past the threshold of 32. Using the static stack (the array) is faster than the dynamic stack as the array is more likely to reside on the same memory page and the risk of a page fault (which would slow down execution) is therefore reduced. As shown by Permandla et al. [41], in a typical application there are no nested critical sections of a depth greater than four. It is therefore highly unlikely that the dynamic stack needs to be used in an ordinary application. The interested reader is advised to read the source code of `inheritance_lock.c` directly.

Algorithm 1 Acquire algorithm called by $thread$ to acquire $lock$

```
1:  function acquire(lock)                          ▷ thread can be obtained through the thread id.
2:      root = rootOf(lock)
3:      if lock.recursive = False ∧thread.inStack(lock) then
4:          return False                            ▷ Prevent acquiring a non-recursive lock more than once.
5:      end if
6:      if lock =Sink then                          ▷ lock is available.
7:          thread.pushOnStack(lock)
8:          lock.parent = thread
9:          return True
10:     else if root = thread then                  ▷ thread has inherited lock.
11:         thread.pushOnStack(lock)
12:         return True
13:     else                                        ▷ thread has to wait for lock.
14:         thread.parent = lock
15:         waitFor(lock)                           ▷ thread will hold lock when this call returns.
16:         return True
17:     end if
18: end function
```

Algorithm 1 shows the semantics of the `acquire`[1] function. $thread$ and $lock$ are ordinary C structures which hold all necessary information. The data of the thread, $thread$, is obtained by using the POSIX function `pthread_getspecific()` which loads thread specific memory of the calling thread. It is therefore vital that pthreads are used for threading when the inheritance lock library is used. The source code of the acquire function can be found in Appendix B.2.

In the first step, the root of the $lock$ node is ascertained (remember threads and locks are nodes in a DAG *forest*). In line 3 it is checked whether the lock is non-recursive (that is, it is not allowed to acquire this lock more than once per thread) and if so, whether the thread has previously acquired this lock. If this is the case the function call was not permitted and therefore returned with $False$. Otherwise there are three possible scenarios as discussed in Chapter 3 (see Figure 3.10). In the first scenario, the lock is available. That is, it does not have a parent and is therefore a *sink*. In the second scenario (line 10) the *root* of the lock's *tree* is the requesting thread itself. The thread has therefore inherited the lock. In the third and final scenario (line 13), the lock is neither available nor inherited by the thread. That means the thread has to wait for the lock to become available. In particular, the lock becomes the thread's parent so that the root of the tree inherits all locks currently held by the thread. As we will see in Algorithm 2, the lock will be automatically assigned to a waiting thread by the thread that is releasing the lock. Therefore, when the call to $waitFor()$ (line 15) returns, the thread will hold the lock.

Algorithm 2 shows the logic of the `release` function. A waiting thread is woken up using a POSIX condition variable. Note that a lock can only be released if it is on top of the thread's lock stack. The source code of the internal release function which corresponds to the pseudo code can also be found in Appendix B.3.

First of all, a lock that is not on top of the thread's stack can not be released (line 2) as discussed in Chapter 3, Subsection 3.4.2 (see Figure 3.12). Otherwise, we can pop the lock from the stack. If the parent node of the lock is not the thread, the thread has inherited the lock and no modifications to the graph should be made (as noted in Subsection 3.4.2). Otherwise we check if there is another thread waiting for the lock (line 9-10). If this is not the case, the lock should be turned back into a sink (no parent) meaning that the lock is available (line 11). If there is another thread waiting for the lock we can turn the waiting thread into a sink (which shows that the

[1]Note: We use the `type set font` to indicate bits from the C source code (Appendix B) and *italics* to indicate bits from pseudo code. However, sometimes the names correspond to both, source and pseudo code, in which case the type setting is chosen by context.

Algorithm 2 Release algorithm called by $thread$ to release $lock$

```
1:  function release(lock)                          ▷ thread can be obtained through the thread id.
2:      if lock ≠ thread.topOfStack() then
3:          return False                            ▷ Only allow release of locks in LIFO fashion.
4:      end if
5:      popStack(thread)
6:      if lock.parent ≠ thread then                ▷ thread has inherited lock
7:          return True
8:      end if
9:      next = lock.dequeueWaitqueue()
10:     if next =NONE then
11:         lock.parent =NONE
12:     else                                        ▷ Another thread is waiting for lock
13:         next.parent =NONE
14:         lock.parent = next
15:         next.pushOnStack(lock)
16:         wakeup(next)
17:     end if
18:     return True
19: end function
```

thread is running, Figure 2.3c), allocate the lock to that thread, push the lock on the stack of the waiting thread and eventually wake the waiting thread up to continue in Algorithm 1, line 15.

In this chapter, we presented an implementation of the inheritance locking protocol using the C programming language. We discussed aims, design decisions, the API and some implementation details of the library using high-level pseudo code. In the next chapter we present two benchmarks that we wrote to evaluate locking protocols. We used this benchmark suite to test the introduced inheritance lock library against other well established locking protocols.

Chapter 6

Benchmarks

In the previous chapter we presented an implementation of inheritance locks as a C library. In this chapter we introduce benchmark tests that we used to evaluate locking protocols against deadlock and starvation as well as testing performance of a protocol.

Correctness and performance testing is of importance, because only protocols that have been shown to be correct and reasonably powerful will be used for production. Typically, established benchmarks from other researchers are used to reliably compare results with the outcome of previous work in the field. Therefore, we intensively researched various benchmark suites, however, we struggled to find a benchmark fit for our purpose. We made out three main issues with existing benchmark suites. Firstly, for many benchmark tests the source code was not freely available or not up to date. Similarly, if the source code was available it was often poorly documented and unclear how to use the benchmark. Finally, many benchmark tests were too specialised for a specific purpose. For example, the Varro benchmark uses model transformations for the test suite [50, 14], the STAMP benchmark is specialised for transactional memory applications [33].

Given these outlined issues, we decided to write our own benchmarks to test the implementation of inheritance locks against other, well established, locking protocols. In particular, we wanted to check whether *deadlock* or *starvation* is possible in a given locking protocol and we wanted to measure performance of a given protocol. The protocols that we wanted to test with our benchmark tests were pthread mutexes, C++ mutexes, *transactional memory* (supported natively in GCC with the `-fgnu-tm` flag) and inheritance locks.

6.1 Micro benchmark

The micro benchmark was written to produce clear and reproducible results. In particular, the benchmark performs two simple performance tests and two further tests for deadlock and starvation of the locking protocol.

6.1.1 Methods and design

Details of the locking protocol had to be given to the benchmark through the `benchmark.h` header file as pre-processor definitions. These definitions were then replaced by the pre-processor in the benchmark source code to perform library calls at the according places. Eventually, an individual executable per protocol was produced by linking the benchmark object file with the according libraries. Besides the obvious calls to the locking library, such as `lock()` and `unlock()`, the threading library calls also had to be defined in the `benchmark.h` header file. This was because certain locking libraries might require a specific threading library in order to work

correctly. For example, inheritance locks only work with pthreads while C++ mutexes should be used with C++ threads.

The tests performed by the benchmark were in this order:

- Performance test of a single thread: locking and afterwards unlocking 10^5 locks in stack fashion.
- Performance test of a single thread: locking and unlocking a single lock 10^5 times.
- Produce a deadlock by making two threads waiting for a lock held by the other one.
- Produce starvation by making three threads compete for two resources, such that the third thread is likely to starve.

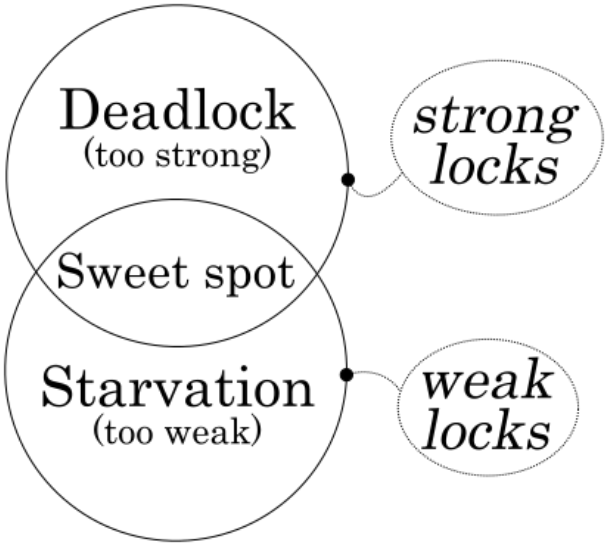


Figure 6.1: Deadlock and starvation are mutually exclusive

The results are given on standard output. If a test fails (in particular tests three and four), the failure is reported and the benchmark aborted. Deadlock occurs if the used locking protocol is too strong, that is locks will in no circumstance be pre-empted. On the other hand, the locking protocol is too weak if starvation occurs, that is locks will always be pre-empted or rolled back and a thread might struggle to make progress. Deadlock and starvation are therefore mutual exclusive as Figure 6.1 shows. If a locking protocol suffers from deadlock, it can not suffer from starvation. However, this is only true in locking protocols which enforce a FIFO wait queue for all threads waiting for a lock. For this reason we ran test four individually if test three had failed for a protocol.

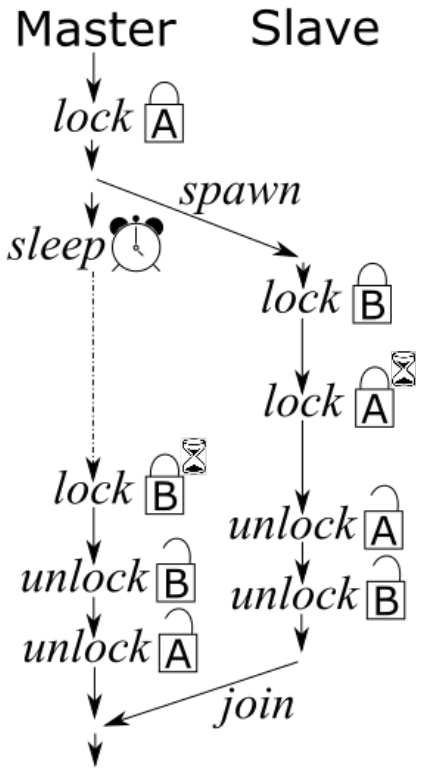


Figure 6.2: Micro Benchmark - Test 3 Deadlock

The first and second tests are self-explanatory. The third test (deadlock) was designed as sketched in Figure 6.2. In particular, if the protocol deadlocks the master and slave threads will be stuck in the states indicated with hourglasses. If the master thread does not return within a predefined timeout, the protocol is assumed to be deadlocked and the benchmark test is aborted.

For the fourth test three threads, Master, Partner and Slave, were created. As shown in Figure 6.3 the Master and Partner threads take indefinitely turns in unlocking and locking lock `A` and `B` respectively. This procedure is only stopped if the Slave thread successfully locks and unlocks both locks at the same time. Note that by design lock `A` and `B` are never available **at the same time**. That means if the protocol can not prevent starvation, thread Slave will never be able to finish (because the thread can not hold and wait) and therefore starve. If the Master thread does not return within a predefined timeout, the protocol is assumed to suffer from starvation and the benchmark test is aborted.

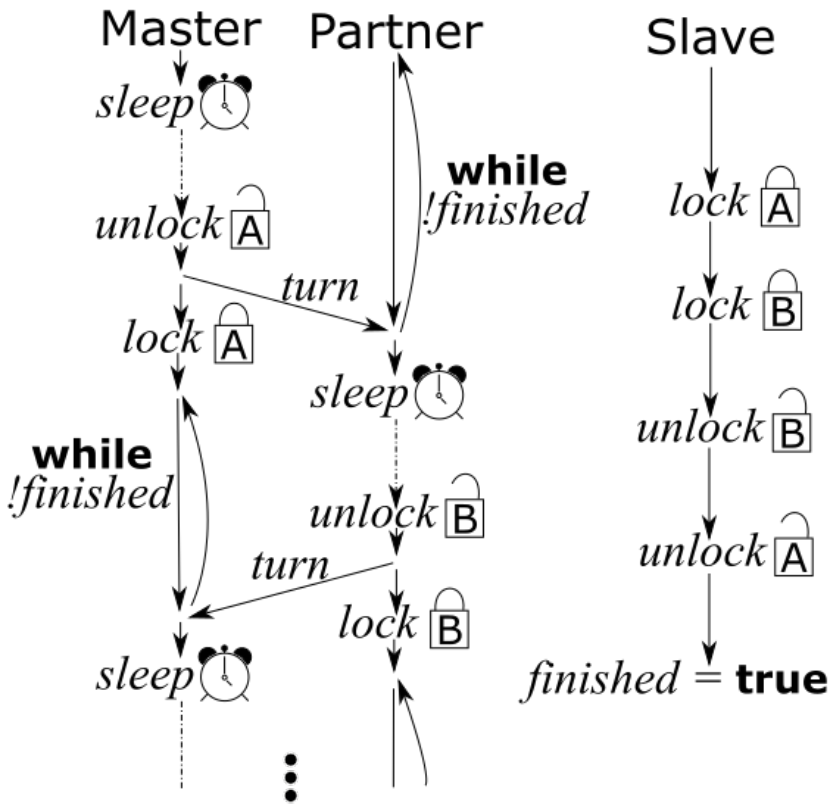


Figure 6.3: Micro Benchmark - Test 4 Starvation

Transactional memory can not be tested with the micro benchmark, because a transactional memory scoped block does not allow side effects inside of it. A side effect is any kind of operation which can not easily be undone or redone. In particular, for the third and fourth test (deadlock and starvation) it was necessary to put threads into sleep for certain times in order to accomplished the wanted scenario. Putting a thread into sleep is considered a side effect and is therefore not allowed in a transactional memory block. To address this issues we wrote the Tyche Benchmark which we introduce in the next section.

The Micro benchmark test was run against pthread mutexes, C++ mutexes and inheritance locks. Each case was run 100 times and the arithmetic mean time was taken (excluding negative times which were occasionally reported). Details of the used machine can be found in Table 4.2. We ran the benchmark tests with both, optimisation disabled and with optimisation enabled using the `-O3` flag.

6.1.2 Results

The test results of the runs with optimisation disabled can be found in Table 6.1 and the results with optimisation enabled, using the `-O3` flag, in Table 6.2.

Table 6.1: Micro benchmark test results (no optimisation)

	pthreads	C++	Inheritance Locks
Test 1 - Average runtime in ms	3.64 ($\sigma = 0.80$)	2.92 ($\sigma = 0.58$)	14.46 ($\sigma = 3.65$)
Test 2 - Average runtime in ms	2.36 ($\sigma = 0.49$)	2.85 ($\sigma = 0.60$)	5.42 ($\sigma = 1.35$)
Test 3 - Free from deadlock	N	N	Y
Test 4 - Free from starvation	N	N	Y

Table 6.2: Micro benchmark test results (optimised with `-O3`)

	pthreads	C++	Inheritance Locks
Test 1 - Average runtime in ms	3.37 ($\sigma = 0.70$)	1.96 ($\sigma = 0.43$)	14.78 ($\sigma = 3.50$)
Test 2 - Average runtime in ms	2.24 ($\sigma = 0.49$)	2.35 ($\sigma = 0.53$)	5.20 ($\sigma = 1.13$)
Test 3 - Free from deadlock	N	N	Y
Test 4 - Free from starvation	N	N	Y

On average the C++ mutexes were the fastest option in the first benchmark test and pthread mutexes the fastest option in the second benchmark test. Inheritance locks were constantly slower than pthread mutexes or C++ mutexes. Pthread and C++ mutexes were found to be prone to deadlock. Furthermore, pthread and C++ mutexes were found to be prone to starvation. This is because waiting threads (i.e. the slave thread) are woken up, that is placed in the ready queue, once the corresponding mutex is unlocked, however, the mutex will be locked on a first come, first served basis [15]. Since the unlocking thread (master or partner) is usually still running, it is highly likely that it will be successful in reacquiring the lock. Hence, the woken thread will fail in acquiring the lock and becomes blocked again; ergo starvation is possible. This recorded behaviour in pthread and C++ mutexes can decrease performance as the convey phenomenon shows [4]. Inheritance locks were found to be free from deadlock and starvation.

6.1.3 Discussion

It was to be expected that C++ and pthread mutexes would deliver similar runtimes. It is also not surprising that inheritance locks were found to be slower as the library requires for each request on average a locking and unlocking request of a pthread mutex, `graphLock`, in order to lock and unlock the *wait-for graph* as discussed in Chapter 5. Pthread mutexes are therefore at least twice as fast as inheritance locks. Additionally to locking and unlocking the `graphLock` for the wait-for graph, the inheritance lock library has to perform a number of protocol specific calculations to ensure correct behaviour. The experienced slowdown is therefore reasonable. In Chapter 8 we discuss options which could speed the inheritance lock library up.

6.2 Tyche benchmark

Since the Micro Benchmark, introduced in the previous section, can not be used with *transactional memory*, we decided to develop another benchmark which firstly, can be run with transactional memory and secondly, allows a high degree of randomisation. The high degree of randomisation was of importance to account for a great variety of scenarios, even though this decreases reproducibility. The name Tyche Benchmark was chosen to reflect this characteristic. Tyche is a patron for fortune, luck and destiny in ancient Greek mythology.

6.2.1 Methods and design

The tyche benchmark uses a header file, `benchmark.h`, to define the locking and threading libraries. This pre-processing approach is equivalent to the micro benchmark as introduced in the first paragraph of Subsection 6.1.1.

Three parameters can be given to the tyche benchmark, a seed for the random function, the maximum number of threads and the maximum number of locks. The seed for the random function is set only once at the beginning of the program. The seed can be given as a program parameter to achieve higher reproducibility. Otherwise,

the current time is used as the default seed (the default should be used for all experiments). The maximum number of threads defines the upper bound of the range of which a random number of threads is chosen from ($numberOfThreads = random(range(1, maxThreads))$). Equally the maximum number of locks defines the upper bound of the random range for the number of locks in the program.

Each lock is embedded in an individual data structure. This data structure consists of the lock, which can be acquired and released by the corresponding locking library, and a data field that is increased by two after each locking request and decreased by one before each unlocking request. The data field is necessary for two reasons. Firstly, it ensures that the transactional memory library behaves in the same way as a conventional locking library (that is, the data is consistent between locking and unlocking) and secondly, it allows to check how often each lock has been acquired by all threads combined at the end of a test.

Collatz Conjecture In the core of the algorithm we make intensive use of *Collatz conjecture*. The conjecture says that, given any positive number greater than one; if the number is divided by two if it is even or multiplied by three and incremented by one if it is odd, the number will become one after repeating this step for a number of times. Formally:

$$f(n) = \begin{cases} 1 & \text{if } n <= 1 \\ n/2 & \text{else if } n \mod 2 \equiv 0 \\ n3+1 & \text{else if } n \mod 2 \equiv 1 \end{cases}$$

Sequence of Collatz conjecture:

$$a_i = \begin{cases} n & \text{for } i = 0 \\ f(a_{i-1}) & \text{for } i > 0 \end{cases}$$

Example. Collatz conjecture of 7:
$7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1$

The spread between even and odd numbers for any given progression is on average even, but the occurrence of even and odd numbers in the sequence is semi-random. The longest progression for any given starting number in a range of numbers is well known [10, 2] (for example: The starting number 9780657631 takes 1132 steps until it reaches one which is the longest progression for any given starting number less than 10 billion).

Algorithm 3 shows how Collatz conjecture is used in the tyche benchmark. Each thread uses a random seed for an individual Collatz conjecture from the range (1, 1000). The longest possible progression in this range is 871 with 178 steps. A thread can therefore lock at most 178 locks before it terminates. If the current number is not divisible by three, a new semi-random lock is locked and added to a local saved (i.e. thread specific) stack (line 4-8). 178 random values are pre-evaluated and delivered in $sRandom()$. This is because the random function has side effects and is therefore not allowed in a transactional block. On average more locks are acquired than released (two out of three times). This greedy behaviour increases competition for locks between threads and makes deadlock and starvation more likely.

We decided to run the tyche benchmark with inheritance locks and transactional memory. We also tried to run the benchmark test against pthread and C++ mutexes, however, as expected they deadlocked immediately. We wanted to test a wide range of parameters. In particular, we chose to run tests against (maximal) 5, 10, 15, 25 and 50 locks. For each of these tests we ran with (maximal) 10 to 100 threads in incremental steps of 10. We ran each particular parameter setting 1000 times. This means we ran in total 10^5 tests (2 locking systems x 5 lock numbers x 10 thread numbers x 1000 runs per setting). Furthermore, we initially experienced performance

Algorithm 3 Core of the Tyche Benchmark

```
1: c = random(range(1, 1000))
2: BEGIN TRANSACTION
3: while c ≠ 1 do
4:     if c not divisible by 3 then
5:         target = sRandom()
6:         stack.push(dataStructure[target])
7:         lock(dataStructure[target].lock)
8:         dataStrucutre[target].data ← dataStructure[target].data + 2
9:     else if stack.notEmpty() then
10:        target = stack.pop()
11:        target.data ← target.data − 1
12:        unlock(target.lock)
13:    end if
14:    c = nextCollatz(c)
15: end while
16: while stack.notEmpty() do
17:    target = stack.pop()
18:    target.data ← target.data − 1
19:    unlock(target.lock)
20: end while
21: END TRANSACTION
```

differences which seemed to depend on processor load, heat and ultimately processing speed. To address this issue we wrote a bash script which evenly fragmented the test runs in order to evenly spread the processor load between the different test runs. Details of the used machine can be found in Table 4.2. Appendix C contains the used scripts. All test were run on executables which were compiled with optimisation enabled (`-O3`).

6.2.2 Results

The total execution time of all benchmark tests combined was just over 26 minutes. The results of the tests are shown in Figure 6.4 using box plots. The box indicates the second and third quartiles separated by the median. The lower and upper whiskers indicate the 2^{nd} and 98^{th} percentiles. In each graph the number of the maximal locks is fixed as indicated. The X-axis indicates the number of maximal threads, the Y-axis indicates the runtime in milliseconds. The blue plots (dashed, left hand side) indicate transactional memory, the red ones (solid, right hand side) inheritance locks.

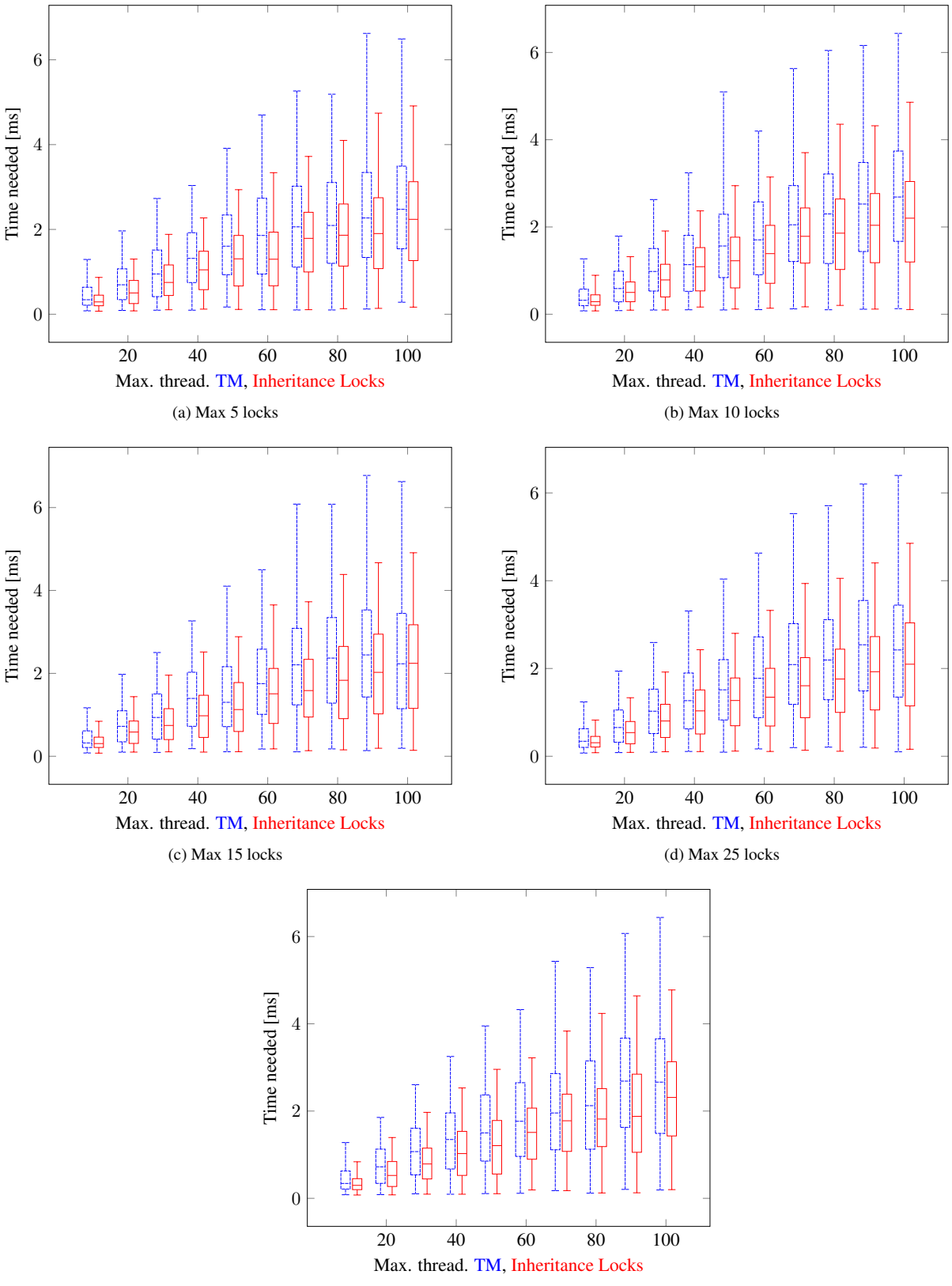
The figures clearly suggest that inheritance locks are on average faster than transactional memory. In the worst case (upper whisker), inheritance locks are always faster than transactional memory. In the best case (lower whisker), inheritance locks and transactional memory are similarly fast. On the other hand, a t-test which was run on a number of the result sets could not determine a difference of statistical significance between the experiment groups.

6.2.3 Discussion

It is difficult to justify the slow down of the integrated GCC transactional memory compared to inheritance locks without deeper knowledge of the transactional memory implementation. However, the most likely reason for the slow down is *starvation* which can occur in transactional memory as pointed out by Herlihy and Moss [19].

The better performance of inheritance locks compared to transactional memory makes inheritance locks a feasible alternative to well established locking protocols.

In this chapter we discussed two benchmarks which we used for evaluation of inheritance locks and for comparison against other well established locking protocols. Our results show that inheritance locks are fit for purpose and achieve partially better results than conventional locking approaches. In the next chapter we present the results of macro-benchmarks, that is, running real world and widely used applications with inheritance locks rather than conventional locks.



(a) Max 5 locks

(b) Max 10 locks

(c) Max 15 locks

(d) Max 25 locks

(e) Max 50 locks

Figure 6.4: Tyche Benchmark - Runtime results

Chapter 7

Application stress test

In the previous chapter we introduce two benchmarks that we used for evaluating inheritance locks and compare various locking protocols. However, the validity of the benchmark results is to some extent limited, because benchmark tests do not necessarily correspond to common usage of a locking protocol. For this reason, we wrote a shared library, `libInheritPthread.so` which is a wrapper library for the inheritance lock library (introduced in Chapter 5). It interposes calls to pthread mutexes and redirects those calls to the inheritance lock library instead. In this chapter we introduce the shared library and give a brief overview of the implementation, we present the results of running the library with a number of popular applications and show the results of a stress test using the shared library and the GNU Image Manipulation Program (GIMP).

7.1 Implementation

We introduced the inheritance locking library in Chapter 5 which is the core of the shared library. Furthermore, it uses a generic map of pthread mutexes to inheritance locks and consists of a number of interposing functions for pthread mutex calls. These interposing functions are implemented in the source file `pthread_interpose.c` (Appendix D.1). In particular, the following function calls are interposed:

- `int pthread_mutex_lock(pthread_mutex_t *mutex)` This call will be forwarded to the pthread_mutex_lock function iff `mutex` is the graph lock of the inheritance locking library. Otherwise, a bijective (one-to-one) mapping is used to determine the inheritance lock which corresponds to the mutex. If no mapping for the mutex has been established a new inheritance lock mapped to the mutex will be created. The `acquire` function of the inheritance locking library is called using the mapped inheritance lock.

- `int pthread_mutex_unlock(pthread_mutex_t *mutex)` This call will be forwarded to the pthread_mutex_unlock function iff `mutex` is the graph lock of the inheritance locking library. Otherwise, the bijective mapping is used to determine the inheritance lock which corresponds to the mutex and the `release` function of the inheritance locking library is called instead.

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)` This call can currently not be mapped to the inheritance locking library and execution is therefore aborted.

- `int pthread_mutex_destroy(pthread_mutex_t *mutex)` Forwards this call to the real implementation of pthread_mutex_destroy and destroys the associated inheritance lock (in case it exists) of the mutex using the bijective mapping.

The functions above can be used as pure forward functions, that is they simply log and forward the request to the pthread function, by setting the `USE_PTHREAD` constant to 1. The function calls are therefore tunnelled to the pthread library (Column 1, Table 7.1).

Furthermore, the library can interpose and report function calls to the pthread condition variable functions `pthread_cond_timedwait()`, `pthread_cond_wait()` and `pthread_cond_signal()`. This behaviour can be enabled by defining the pre-processor global `COND`. It is important to interpose these function calls as well, because there is currently no corresponding implementation in the inheritance locking library (as discussed in Chapter 8). This means, the usage of condition variables will currently lead to execution failure if the interposing inheritance locking library is used.

Last but not least, we wrote a shared stub library which simply answered all of the function calls above positively without any locking. This stub library is a useful tool to decide whether performed locking calls of the application are actually necessary (Column 3, Table 7.1).

7.2 Application testing

As discussed, using benchmarks for the evaluation of a new locking protocol, can produce non-representative results and does not necessarily reflect real scenarios in which the locking protocol would be used. It is therefore desirable to test the locking protocol directly with popular applications.

7.2.1 Design

We ran a number of popular applications with various locking libraries which we put in place using the `LD_PRELOAD=` dynamic linker facility, which allows to load custom shared libraries before any other dynamic library. This means that the custom library effectively overwrites all functions with an identical signature in the later loaded dynamic libraries. We tested correct start up, observational correct behaviour and correct closing. We tested the inheritance lock library (once with `USE_PTHREAD = 1` and once with `USE_PTHREAD = 0`) as well as the stub library. We ran each test three times.

7.2.2 Results

Table 7.1 shows the results of this experiment. Red cells indicate a failure of the application using the corresponding locking library. The cell content gives further information about the nature of the failure.

Table 7.1: LD_PRELOAD Dynamic shared object library with popular applications

Application	Tunnelled Pthreads	Inheritance Locks	Stub Library
gnome-calculator	Y	Y	Y
libreoffice	Y	Waits for condition variable	Undefined Behaviour
firefox	Core dumps	Core dumps	Core dumps
gedit	Y	Y	Y
nautilus	Y	Y	Y
inkscape	Y	Y	Y
gnote	Y	Y	Y
gimp	Y	Y	Y
google-chrome	Y	Releases lock which is not on top of stack	Core dumps

As we can see, most applications did not struggle with any given locking library even if the library, as it is the case in the stub library, does not perform any useful operation. Moreover, it is worth noting that the firefox browser can not be started with LD_PRELOAD in place. Further investigations revealed that even an empty shared object library (i.e. which does not overwrite any functions) causes firefox to crash (core dumped) if it is included with the LD_PRELOAD directive. For this reason we ignore the results produced by firefox as the preloaded library itself is clearly not the origin of the failure.

7.2.3 Conclusion

From these results we conclude that inheritance locks work well with ordinary applications and that the causes of failure can be traced back to compatibility issues between pthread mutexes and inheritance locks. On the other hand, based on the results obtained from the stub library, our results suggest that locking libraries are often conservatively employed for worst-case, highly contended resource usage. However, this is rare in general application scenarios. Similar results have been found by Rajwar and Goodman during their work on speculative lock elision [42].

7.3 Stress test

Even though the results from the previous test give a good indication of correct application behaviour with alternative locking protocols in place, they are not resilient as they are purely subjective and only tested a limited number of times. The former is of significance, because it is difficult to judge from the graphical user interface (GUI) alone whether an application behaves correctly or not. The latter is of particular importance, because multi-threaded applications can cause errors which are difficult to reproduce and only depend on the scheduling order of the involved threads. We could observe such an error-prone behaviour with the libreoffice application in combination with the stub library (see Table 7.1). At times the application worked, while it crashed other times. This can be described as undefined behaviour.

7.3.1 Design

Given theses limitations, we deduce that the results from the previous test are insufficient for a reliable conclusion. Therefore we decided to use the GNU Image Manipulation Program, GIMP, as the basis for an application stress test. In particular, we made use of GIMP's batch processing capability which allows to process an image file with a pre-written script from the command line interface. This makes it suitable for writing a bash script for a stress test evaluation while the GUI is avoided altogether (Appendix D.2). We used the Wikipedia logo, saved as a PNG file, as the image file for the stress test (file details can be found in Table 7.2). The batch script applied the GIMP unsharp-mask plugin to the file (Appendix D.3). We created a reference file by applying the batch script without a pre-loaded library. We could then use this reference file to perform a byte-wise comparison with a file that has been processed with a different locking library in place. We had to skip the first 100 bytes in the byte-wise comparison as this range contains meta-data including a time stamp which would differ between files. We ran the test 500 times with the inheritance lock library, the stub library and, as a control group, without a preloaded library.

Table 7.2: Image file for the stress test

Content	Wikipedia Logo
Type	PNG
Dimensions	1058x1058
Bit depth	32
Size	234KB

7.3.2 Results

Table 7.3: Stress test results

Library	Successful runs out of 500 attempts
Inheritance Locks	500
Stub Library	500
Control group	500

Table 7.1 shows the results of the application stress test. We did not record any failures in any of the three test groups.

7.3.3 Conclusion

These results lend weight to our previous findings, namely, inheritance locks work well in a real application and locking is often done where it is not necessary.

7.4 Discussion

The results of these tests allow an insight into resource locking of popular applications. Our findings suggest that locking is often performed in places where this is not necessary therefore posing an unnecessary performance overhead. On the other hand, further implementation details of each application are required in order to effectively support this hypothesis.

Moreover, our results suggest that inheritance locks work well with many popular applications, however, more work needs to be done in order to increase compatibility of the inheritance lock library as discussed in the next chapter.

In this chapter we introduced the results of running the inheritance lock library with popular applications using the `LD_PRELOAD` directive. Our results suggest that often unnecessary locking operations are performed in applications and that more work needs to be done in order to improve compatibility of the inheritance lock library. In the next chapter we will discuss these compatibility issues in depth and illustrated possible solutions in order to tackle these limitations.

Chapter 8

Scenarios & Limitations

In the previous chapter we presented results of running the inheritance lock library with popular applications. In this context, we observed some limitations of the current library implementation which we will discuss in more depth over the course of this chapter.

In this chapter we first revisit the dining philosophers problem and introduce an example scenario in which inheritance locks are particularly useful. Then, we focus on some of the limitations of the current library implementation which is strongly connected to the previous chapter. We also consider approaches to eliminate these limitations and eventually discuss condition variables in inheritance locks.

8.1 Scenarios

8.1.1 Dining Philosophers with Inheritance Locks

In Subsection 2.2.3 we introduced the *dining philosophers problem*. In this subsection we consider what happens when inheritance locks are used to guard the forks in the problem. Generally, no philosopher would ever lend a fork that he is already holding. The only exception to this is a situation as shown in Figure 8.1. Philosopher five inherits the right fork of philosopher one as everyone is waiting for philosopher five. Once philosopher five has finished eating, philosophers four, three, two and one will eat before five can eat again. This is because as soon as five finishes eating, philosophers one and four will take over five's forks meaning that four can start eating. This eating behaviour then goes round the table. In other words, once four finishes eating three can start eating, once three finishes eating two can start eating and, finally, once two finishes eating one can start eating. Because of the round going eating it is impossible that all four philosophers are waiting for five, ergo one will not lend five his fork again until he has eaten. This means, philosopher one (in fact, any philosopher at the table) will not starve, because he is guaranteed to always eventually eat.

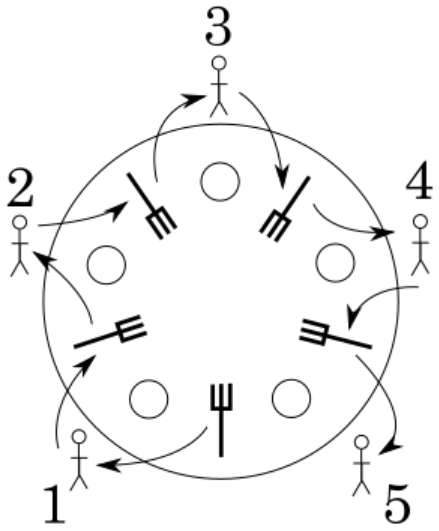


Figure 8.1: Philosopher 1 will lend Philosopher 5 his right fork.

8.1.2 Bank Transaction

In this subsection we present an example scenario in which inheritance locks are particularly useful compared to mutually exclusive locking. In particular, consider a banking system with no allowed overdraft (for simplicity). If we want to transfer money from account A to B we would first lock both accounts so that no other process can interfere with our transaction. Then we would check if account A has sufficient funds for the transaction and if so we would reduce account A's balance by the amount and increase account B's balance accordingly. Eventually we would unlock both accounts. Algorithm 4 illustrates the semantics of the transfer logic.

Algorithm 4 Transfer money between bank accounts

```
1:  function transfer(amount, sender, receiver)      ▷ Transfer amount from sender to receiver
2:      lockAccount(sender)
3:      lockAccount(receiver)
4:      if sender.balance >= amount then             ▷ Avoid overdraft
5:          sender.balance− = amount
6:          receiver.balance+ = amount
7:      end if
8:      unlockAccount(receiver)
9:      unlockAccount(sender)
10: end function
```

Let us now consider a process, P_1, which wants to transfer an arbitrary amount from account A to B. P_1 calls the transfer function and locks the bank account A. Let us now assume that user input has occurred and P_1 is therefore interrupted. Particularly, the user wishes to run process P_2 which transfers an amount from account B to A. The Kernel therefore spawns P_2 which becomes blocked while attempting to lock account A, because A has already been locked by P_1. Since P_2 is blocked, control is eventually returned to P_1 which equally can not continue, because B has been locked by P_2 (see Figure 8.2a). That means the system is *deadlocked*. On the other hand, if inheritance locks are used, P_1 could continue and successfully lock account B, because B is held by P_2 which is waiting for A which is held by P_1. P_1 is therefore the *root* of resource B (Figure 8.2b).

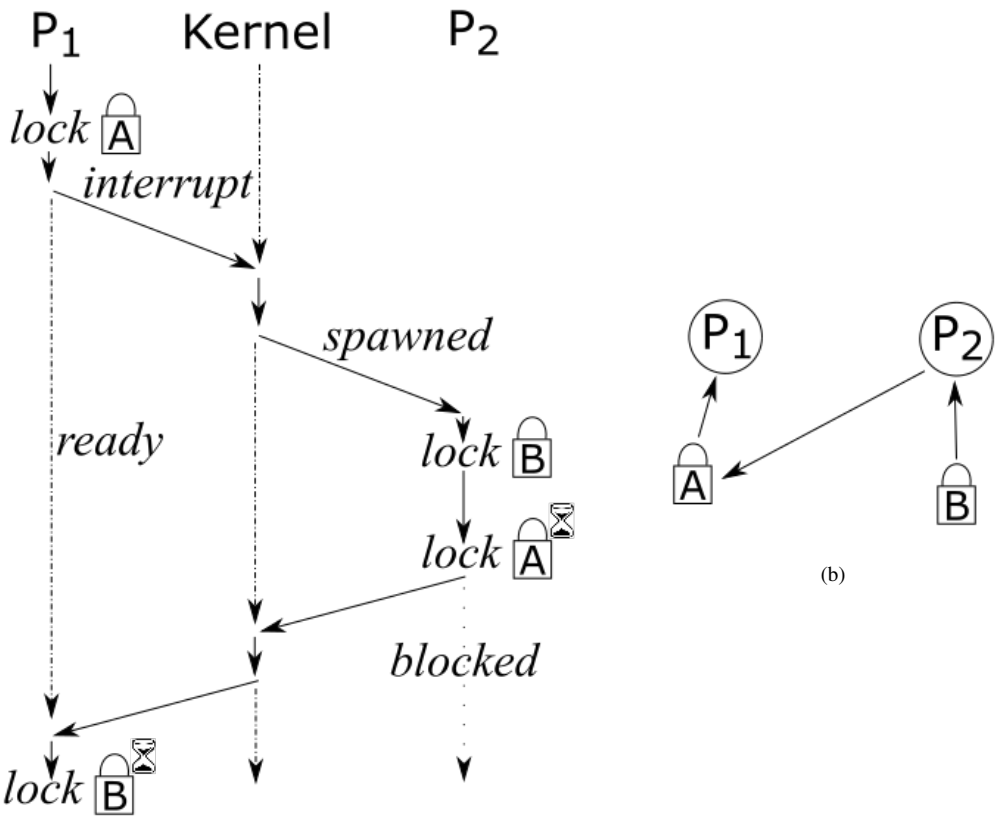


Figure 8.2: Example Scenario - Bank transfers

8.2 Limitations

As we can see in the previous scenario, inheritance locks have many advantages over ordinary mutexes in particular if they are used in generic situations, that is, it is not clear in advance which particular concrete lock will be locked and unlocked. On the other hand, there are also a number of drawbacks if inheritance locks are used. In this section we will discuss those limitations and risks of inheritance locks.

First of all, there is a risk that a locked resource changes if another lock is acquired. For example, Algorithm 5 might be considered more efficient than Algorithm 4 from the previous section. In terms of efficiency, this statement is certainly true. Why bother locking the receiver if the sender has insufficient funds for the transaction?

Algorithm 5 Transfer money between bank accounts

```
1:  function transfer(amount, sender, receiver)      ▷ Transfer amount from sender to receiver
2:      lockAccount(sender)
3:      if sender.balance >= amount then             ▷ Avoid overdraft
4:          lockAccount(receiver)
5:          sender.balance− = amount
6:          receiver.balance+ = amount
7:          unlockAccount(receiver)
8:      end if
9:      unlockAccount(sender)
10: end function
```

Using Algorithm 5 is problematic with inheritance locks, though. This is because in line 4, when we try to acquire the receiver, we might temporarily lend the sender to a different process meaning that the senders balance might have changed and $sender.balance >= amount$ (line 3) is not true any more.

A further limitation of inheritance locks is the requirement to lock and unlock locks in a stack fashion (LIFO). In the previous chapter we observed that not all applications lock and unlock in such a stack fashion which was to be expected. In particular, the execution of Google Chrome failed because the application was attempting to release a lock which was not on top of the thread's stack (Subsection 7.2.2).

Finally, condition variables are currently not supported by inheritance locks. Many applications use condition variables to wait for a condition to occur or to alert waiting threads. In the previous chapter we saw that libreoffice uses condition variables which meant that the execution had to be aborted when used in combination with the inheritance lock library (Subsection 7.2.2).

8.3 Library usage

In this section we present a brief user manual and guidelines for using the inheritance lock library.

As a rule of thumb, the programmer should always put much effort into releasing locks in the reversed order they were acquired. If a lock is acquired in a function, it should be released before the function returns. Otherwise, it could be extremely difficult to guarantee correct behaviour (i.e. locking and unlocking in LIFO fashion) or to debug code. This is similar to a `synchronized` statement in Java [39].

Let us assume that the previous paragraph is always true. If a function of a library, which uses inheritance locks internally, is called, the state of previously locked resources is guaranteed to stay atomic. This is because the called library can not know about the user code and it is therefore impossible that the library will amend any of the locked resources. Even if the thread should become blocked during the library call, the resources are guaranteed not to be inherited because the thread is blocked while waiting for an internal inheritance lock of the library which can itself only be held by a library function call.

Similarly like the previous paragraph, the programmer can introduce different levels of abstractions for locks and functions. Functions of a certain abstraction level only acquire locks of the same level and only call functions of the same or of a higher level (Figure 8.3). This effectively means that some locks have a different priority than others guaranteeing that only functions using locks of the same priority/level pre-empt (that is inherit) and change resources. This approach greatly simplifies some of the complications coming with inheritance locks. Moreover, it is closely linked to Havender's Approach 1 [18] better known as the resource hierarchy protocol [8, 35].

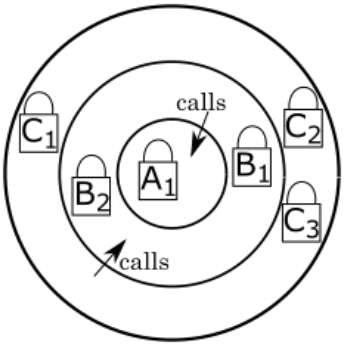


Figure 8.3: Each layer only knows about its own locks and functions as well as higher level functions.

8.4 Possible improvements to the library

We determined a number of limitations regarding inheritance locks such as not being able to release a lock that is not on top of the stack. A very easy solution to this limitation would be the following: Have a flag on each stack element which indicates whether the element should be released once it is on top of the stack (the *release flag*). The `acquire` function would simply check whether the lock is already on the stack (starting from the top to effectively handle recursive locks), if the lock is found and the *release flag* is set, unset the flag. If the flag is not set and it is a recursive lock or if it was not found in the stack at all, push it on top of the stack and leave the found entry as it is. Otherwise, report an error.

Similarly, the `release` function would simply find the lock in the stack (starting from the top) sets the *release flag* or continue the search if the flag is already set. Report an error if no such element has been found. Afterwards keep popping the stack until the top element is not set for release.

A further step would be allowing releasing a lock, l, from the middle of the stack. In particular, this could be done iff l is not a *conveyor resource* for any later acquired locks. That means, if none of the locks in the stack above l have been inherited through l, it is safe to release l (Subsection: 3.4.2, see also Figure 8.4).

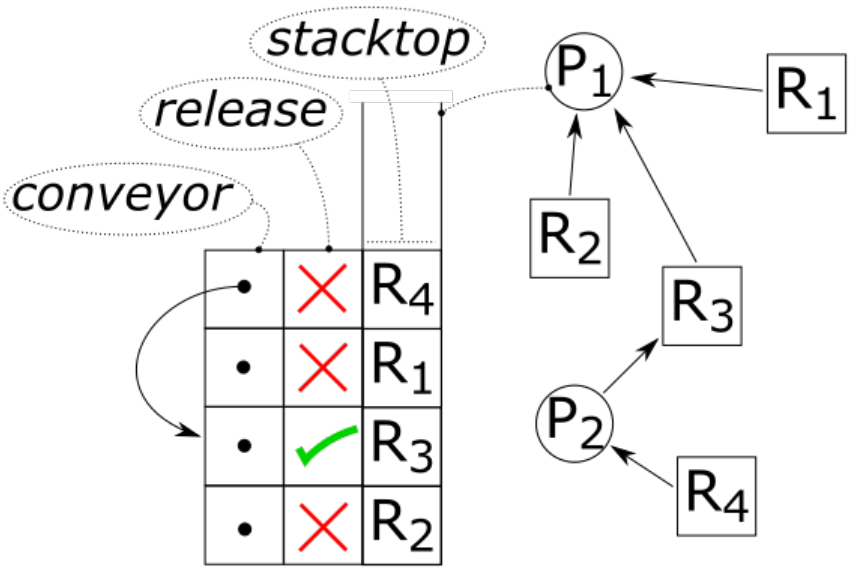


Figure 8.4: R_3 can not be release yet, because it is a conveyor resource for R_4.

Another improvement to the library would be the employment of a spin-lock (with a test-and-set atomic operation) for the *wait-for graph* rather than a pthread mutex because all computations on the graph are quick. Therefore a *busy-wait* (while waiting for the spin-lock) would not introduce a great overhead. As we have seen in the micro benchmark test (Chapter 6) the `graphLock` mutex, acting as the lock for the wait-for graph, is a significant bottleneck for inheritance locks. Based on the obtained results we prognosticate that speed-ups of up to 85% could be achieved if the mutex would be replaced with a spin-lock. On the other hand, a different way to suspend and wake up threads would need to be implemented because the pthread condition variable requires a locked pthread mutex in order to work.

Finally, we observed *starvation* in pthread mutexes (Subsection 6.1.2). This means that, in theory, a thread could struggle to acquire the `graphLock` of the inheritance lock library because at each attempt a different thread has already acquired the mutex and the thread is consequently put to sleep. Even though, we did not experience this behaviour directly, it is likely to occur in larger systems (with several hundred cores) and therefore

posses yet another disadvantage of using pthread mutexes internally. Spin-locks can not entirely remove this risk, however, the risk would be greatly reduced because a calling thread would keep constantly trying to lock the graph rather than going to sleep.

8.5 Condition variables in inheritance locks[1]

A further limitation of inheritance locks is the missing functionality of condition variables. We observed this limitation in the previous chapter (Subsection 7.2.2). A straightforward implementation does not work, because a process which is waiting for a condition variable might still hold a number of resources. If now the process which could signal the condition variable has to run a *critical section* for which a resource is required that is held by the waiting process, a *deadlock* can occur. Similarly, if the waiting process is forced to release and re-acquire all its currently held resource, *starvation* can occur.

While recent work from Agarwal et al. [1] and Joshi et al. [24] show that deadlock detection is possible when locks and condition variables are used, it remains unclear whether it is possible to prevent deadlock and starvation with condition variables in place. This question is therefore of great interest, not just for inheritance locks, but for the wider field in general.

We suggest to study this issue by introducing a third node type in the wait-for graph representing condition variables. This node belongs to a resource node (a host node) and a process might wait for such a condition variable (see Figure 8.5).

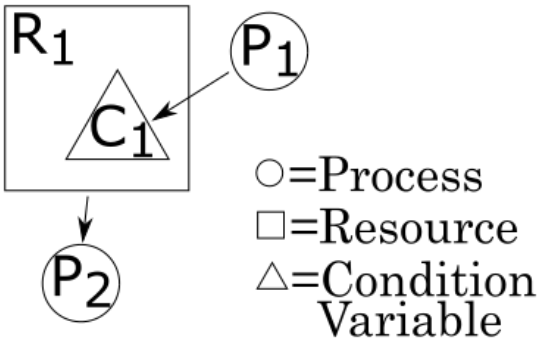


Figure 8.5: P_1 waits for C_1 while P_2 holds R_1.

8.6 Priority inversion

A further widely studied problem regarding locking protocols is the priority inversion problem which was first explored by Lampson and Redell [26]. The problem occurs if a low priority process is holding a resource required by a high priority process. Since the high priority process has to wait for a low priority process, it is effectively graded down to be a low priority process. A common solution for this problem is the priority inheritance protocol introduced by Sha et al. [45]. In a nutshell, a high priority process that is waiting for a low priority process is lending its higher priority to the low priority process until the resource has been released. It is easy to see that the inheritance lock protocol is closely related to the priority inheritance protocol. It should

[1] In this and the following sections we return to the process and resource terminology as more generic aspects of inheritance locking are discussed

therefore be straightforward to improve the inheritance lock library with priority inheritance making it a more powerful library.

In this chapter we revisited the dining philosophers problem and presented a bank transaction scenario using inheritance locks. We discussed limitations of inheritance locks and thoroughly discussed possibilities to reduce those limitations as well as further improve the inheritance lock library. In the next chapter we will conclude and revise the outcomes of this project.

Chapter 9

Conclusion

In the previous chapter we discussed some scenarios in which inheritance locks would be particularly useful as well as thoroughly discussed limitations of the inheritance lock library and possibilities to improve the library. In this chapter we conclude the results of the project as well as discuss future work and learned lessons.

9.1 Summary

The risk of *deadlock*, *starvation* and/or *livelock* is present in many parallel systems and is often ignored. Almost all popular locking protocols do not prevent deadlock, starvation or livelock. If this property is required, for example in a safety critical system, the programmer is required to check for this property by other means for example with a state model checker. This is not ideal especially with the significant increase of concurrent applications on highly connected, parallel systems. Therefore we proposed a novel locking protocol, inheritance locks, which is a refinement of an approach suggest by Havender [18]. While Havender's approach prevents deadlocks it is prone to starvation.

We have shown that inheritance locks prevent starvation by allowing resource pre-emption, but only in certain circumstances. In particular, we thoroughly discussed the origin and the approach of inheritance locks. First we introduced inheritance locks using an illustrative example. Secondly, we gave a precise, mathematical description of inheritance locks using graph theory.

Using the *SPIN* model checker, we were able to verify our claims for a number of resource/process correlations. While we were not able to verify all attempted models, all models were attempted to be verified at least once and each finished verification was positive. These results strongly support our hypothesis.

We implemented inheritance locks as a C library and justified design decisions. We presented the API as well as implementation details using high-level pseudo code.

Using two benchmarks, the micro and the tyche benchmark, we were able to compare the inheritance lock library with other well established and popular locking protocols. Our results show that inheritance locks can be on average at least as fast as *transactional memory* and that inheritance locks, other than pthread and C++ mutexes, are not prone to deadlock and starvation.

Furthermore we ran a number of applications with inheritance locks using the `LD_PRELOAD` directive as well as an application stress test with GIMP. Here, the results suggest that locking libraries are often conservatively employed for worst-case, highly contended resource usage. In other words, locking libraries are often overused.

Finally, we discussed a number of scenarios in which inheritance locks are particularly useful. We thoroughly and critically analysed limitations of the inheritance lock library as well as discussed solution approaches for these limitations.

9.2 Future Work

Even though we believe that the introduction of inheritance locks made a significant advancement to the field, there is still more development and research required in order to improve the inheritance lock library and to fully understand the nature of the problem. In this section we briefly describe some of the possible areas of interest for future work.

- Release locks if they are not on top of the stack

  As we saw in Chapter 7 and discussed in 8.4 the requirement to acquire and release locks in a stack-fashion (LIFO) is a clear disadvantage of the library in terms of usability. We discussed ways how this requirement could be loosened and therefore strongly improve the inheritance lock library.

- Deadlock prevention with condition variables

  To the best of our knowledge, no effective *deadlock prevention* protocol has been introduced that can handle locks, condition variables and semaphores. In 8.5 we briefly discussed a first step towards condition variables in the inheritance lock library. However, much more research in this area is required to fully understand the issue.

- Use a spin-lock to guard the wait-for graph

  As discussed in 8.4 the *wait-for graph* is currently locked by a pthread mutex in the inheritance lock library. This is not ideal because locking and unlocking the pthread mutex is a significant bottleneck of the library. Furthermore, threads become blocked and put to sleep when they have to wait for a pthread mutex to become available. In the worst case this might lead to starvation because a ptread mutex does not have a FIFO wait queue. Moreover, operations on the wait-for graph are usually fairly quick and putting a waiting thread to sleep slows down execution by introducing avoidable context switches. We therefore suggest replacing the `graphLock` mutex with a spin-lock in the inheritance lock library.

- Priority Inheritance

  The priority inheritance protocol is closely related to inheritance locks as we have pointed out in 8.6. Therefore we argue that it is fairly easy to implement priority inheritance in the inheritance lock library. This would form a further significant advantage of the library as priority inversion would be prevented.

9.3 Personal Reflection

The project's idea occurred to me while I was learning about several deadlock prevention protocols and I realised that most of them are prone to starvation. However, at the beginning of this project I had strong doubts about the uniqueness and relevance of the self-proposed topic, Inheritance Locks. Nevertheless, I decided to be not prejudiced against my own idea and pursue the topic further.

During the project I learned many research skills and it trained me to be critical and yet fair to my own and other's ideas. Apart from literature review skills, I also improved my academic writing skills. In terms of programming, I particularly improved my scripting skills specifically shell scripting.

I am pleased with the outcomes of the project and I am glad to say that pursuing a self-proposed topic was a great decision. I was impressed by some of the obtained results and would use inheritance locks in my own programming projects.

9.4 Acknowledgements

I would like to thank my supervisor Dr Jeremy Singer and Anna Lito Michala for their great support and assistance. Their feedback was often critical, but always fair and very helpful.

Bibliography

[1] Rahul Agarwal and Scott D Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 51–60. ACM, 2006.

[2] Ştefan Andrei and Cristian Masalagiu. About the collatz conjecture. *Acta Informatica*, 35(2):167–179, 1998.

[3] Valmir C Barbosa. The combinatorics of resource sharing. In *Models for Parallel and Distributed Computation*, pages 27–52. Springer, 2002.

[4] Mike Blasgen, Jim Gray, Mike Mitoma, and Tom Price. The convoy phenomenon. *ACM SIGOPS Operating Systems Review*, 13(2):20–25, 1979.

[5] Edward G Coffman, Melanie Elphick, and Arie Shoshani. System deadlocks. *ACM Computing Surveys (CSUR)*, 3(2):67–78, 1971.

[6] Ajoy Kumar Datta, Ramesh Dutt Javagal, and Sukumar Ghosh. An algorithm for preventing deadlocks in distributed systems. In *Computers and Communications, 1992. Conference Proceedings., Eleventh Annual International Phoenix Conference on*, pages 109–116. IEEE, 1992.

[7] Edsger W. Dijkstra. De bankiers algorithme. Banker's Algorithm, 1965.

[8] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. *Acta informatica*, 1(2):115–138, 1971.

[9] Edsger W. Dijkstra. Ewd manuscript 1000, 1987.

[10] Shalom Eliahou. The 3x+ 1 problem: new lower bounds on nontrivial cycle lengths. *Discrete mathematics*, 118(1-3):45–56, 1993.

[11] Joaquin Ezpeleta, Jose Manuel Colom, and Javier Martinez. A petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE transactions on robotics and automation*, 11(2):173–184, 1995.

[12] Luca Ferrarini, Luigi Piroddi, and Stefano Allegri. A comparative performance analysis of deadlock avoidance control algorithms for fms. *Journal of Intelligent Manufacturing*, 10(6):569–585, 1999.

[13] GCC. Transactional memory in gcc. `https://gcc.gnu.org/wiki/TransactionalMemory`, 2012. Accessed on: 02/02/2017.

[14] Rubino Geiß and Moritz Kroll. On improvements of the varro benchmark for graph transformation tools. *Universität Karlsruhe, IPD Goos, Tech. Rep*, 7(12), 2007.

[15] GNU. Source code pthread_mutex_lock(). `http://code.metager.de/source/xref/gnu/glibc/nptl/pthread_mutex_lock.c`, 2002. Accessed on: 15/03/2017.

[16] A Nico Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–ff, 1969.

[17] Vicky Hartonas-Garmhausen, Sergio Campos, Alessandro Cimatti, Edmund Clarke, and Fausto Giunchiglia. Verification of a safety-critical railway interlocking system with real-time constraints. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 458–463. IEEE, 1998.

[18] James W. Havender. Avoiding deadlock in multitasking systems. *IBM systems journal*, 7(2):74–84, 1968.

[19] Maurice Herlihy and J Eliot B Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[20] Richard C Holt. Some deadlock properties of computer systems. *ACM Computing Surveys (CSUR)*, 4(3):179–196, 1972.

[21] Gerard Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[22] IBM. Deadlock when eeh frozen and setting multicast. `https://www-304.ibm.com/support/docview.wss?uid=isg1IY60280`, 2004. Accessed on: 08/02/2017.

[23] Sreekaanth S Isloor and T Anthony Marsland. The deadlock problem: An overview. *Computer*, 13(9):58–78, 1980.

[24] Pallavi Joshi, Mayur Naik, Koushik Sen, and David Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 327–336. ACM, 2010.

[25] Wikimedia Kopiersperre. Graph of "internet hosts count 1964-2018, logarithmic scale" based on data from the internet systems consortium. `https://commons.wikimedia.org/wiki/File:Internet_Hosts_Count_log.svg`, 2014. Accessed on: 08/02/2017.

[26] Butler W Lampson and David D Redell. Experience with processes and monitors in mesa. *Communications of the ACM*, 23(2):105–117, 1980.

[27] Gertrude Neuman Levine. The classification of deadlock prevention and avoidance is erroneous. *ACM SIGOPS Operating Systems Review*, 39(2):47–50, 2005.

[28] ZhiWu Li and MengChu Zhou. Elementary siphons of petri nets and their application to deadlock prevention in flexible manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 34(1):38–51, 2004.

[29] Don Libes. *Exploring Expect: a Tcl-based toolkit for automating interactive programs*. "O'Reilly Media, Inc.", 1995.

[30] Lin Lou, Feilong Tang, Ilsun You, Minyi Guo, Yao Shen, and Li Li. An effective deadlock prevention mechanism for distributed transaction management. In *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2011 Fifth International Conference on*, pages 120–127. IEEE, 2011.

[31] Daniel A Menasce and Richard R Muntz. Locking and deadlock detection in distributed data bases. *IEEE Transactions on Software Engineering*, (3):195–202, 1979.

[32] Microsoft. Wait chain traversal. `https://msdn.microsoft.com/en-us/library/windows/desktop/ms681622(v=vs.85).aspx`, 2017. Accessed on: 02/02/2017.

[33] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. Stamp: Stanford transactional applications for multi-processing. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 35–46. IEEE, 2008.

[34] Toshimi Minoura. Deadlock avoidance revisited. *Journal of the ACM (JACM)*, 29(4):1023–1048, 1982.

[35] Waleed A. Muhanna. Composite programs: Hierarchical construction, circularity, and deadlocks. *IEEE transactions on software engineering*, 17(4):320–333, 1991.

[36] Mohamed Naimi, Michel Trehel, and André Arnold. A log (n) distributed mutual exclusion algorithm based on path reversal. *Journal of parallel and distributed computing*, 34(1):1–13, 1996.

[37] Fabiano de S Oliveira and Valmir C Barbosa. Revisiting deadlock prevention: A probabilistic approach. *Networks*, 63(2):203–210, 2014.

[38] Oracle. Deadlock status - enterprise manager. `https://docs.oracle.com/cd/B16240_01/doc/doc.102/e16282/oracle_database_help/oracle_database_adralertlogincidenterrorstatus_deadlockerrors.html`, 2009. Accessed on: 08/02/2017.

[39] Oracle. Intrinsic locks and synchronization. `https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html`, 2015. Accessed on: 17/03/2017.

[40] Andrew Oram, Greg Wilson, and Simon Peyton Jones. *Beautiful code*. O'reilly, 2007.

[41] Pratibha Permandla, Michael Roberson, and Chandrasekhar Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language: 1. In *ACM SIGPLAN Notices*, volume 42, page 10. ACM, 2007.

[42] Ravi Rajwar and James R Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 294–305. IEEE Computer Society, 2001.

[43] David J. Rypka and Anthony P. Lucido. Deadlock detection and avoidance for shared logical resources. *IEEE Transactions on Software Engineering*, (5):465–471, 1979.

[44] Martin Schindewolf, Albert Cohen, Wolfgang Karl, Andrea Marongiu, and Luca Benini. Towards transactional memory support for gcc. In *1st GCC Research Opportunities Workshop*, 2009.

[45] Lui Sha, Ragunathan Rajkumar, and John P Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[46] Arie Shoshani and EG Coffman. Sequencing tasks in multiprocess systems to avoid deadlocks. In *Switching and Automata Theory, 1970., IEEE Conference Record of 11th Annual Symposium on*, pages 225–235. IEEE, 1970.

[47] Abraham Silberschatz, Peter B Galvin, Greg Gagne, and A Silberschatz. *Operating system concepts*, volume 4. Addison-wesley Reading, 1998.

[48] Abraham Silberschatz and ZM Kedam. A family of locking protocols for database systems that are modeled by directed graphs. *IEEE Transactions on Software Engineering*, (6):558–562, 1982.

[49] TIOBE. Tiobe index. `http://www.tiobe.com/tiobe-index/`, 2017. Accessed on: 20/02/2017.

[50] Gergely Varro, Andy Schurr, and Daniel Varro. Benchmarking for graph transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05)*, pages 79–88. IEEE, 2005.

[51] Veritas. I/o hang due to a vxvm deadlock in the kblockd context on the linux platform. `https://www.veritas.com/support/en_US/article.TECH68863`, 2015. Accessed on: 08/02/2017.

[52] Amy Williams, William Thies, and Michael D Ernst. Static deadlock detection for java libraries. In *European Conference on Object-Oriented Programming*, pages 602–629. Springer, 2005.

[53] Dieter Zöbel. The deadlock problem: A classifying bibliography. *ACM SIGOPS Operating Systems Review*, 17(4):6–15, 1983.

Appendices

Appendix A

Scripts used for running SPIN in Computer Cluster

A.1 login.exp

```
#!/usr/bin/expect

set timeout 3
set target [lindex $argv 0]
set password [lindex $argv 1]
set locks [lindex $argv 2]

spawn ssh 2079884f@$target
expect {
  "*fingerprint*" {
    send "yes\r"
    exp_continue
  }
  "password" {
    send "$password\r"
    expect {
      "*School of Computing Science*" {
        expect "*bash*"
        send "cd lockingl4project/modelChecker/\r"
        expect "*bash*"
        send "nohup ./remoteWorker.sh $locks $target >/dev/null 2>&1 &\r"
        expect "*bash*"
        send "exit\r"
        expect "*logout*"
        expect eof
        exit 1
      }
      "Permission denied*" {
        send \003
      }
    }
  }
  "*Could not resolve hostname*" {}
}
expect eof
exit 0
```

A.2 remoteWorker.sh

```
#!/bin/bash

#
```

```
# First argument:  Number of locks
# Second argument: Name of the PC
#
# Always check for 2-10 threads (one thread crashes)
#
# Every PC checks first for 2-10 threads
# on specified number of locks
#
# Then all PCs check for 2-3 lock
# with 2-10 threads
#

l=2  #Lower number of threads
u=10 #Upper number of threads
t=3  #Common number of locks

mkdir ./remoteArchive/$1.$2
cp ./Makefile ./remoteArchive/$1.$2/
cp ./creator ./remoteArchive/$1.$2/
cd ./remoteArchive/$1.$2/

for i in `seq $l $u`;
do
  ./creator $i $1 > model.pml
  make verify > $i.$1.$2.txt
done

for i in `seq $l $u`;
do
  for j in `seq 1 $t`;
  do
    ./creator $i $j > model.pml
    make verify > $i.$j.$2.txt
  done
done

rm ./model*
rm ./creator
rm ./Makefile
```

A.3 runExpect.sh

```
#!/bin/bash

# First argument:  Password
# Second argument: Room
# Third argument:  Limit

ROUND=1 # 3 Rounds Default:1
COUNT=4 # Assign 4 to 10 Default:4

strZero="0"
strDash="-"
strSuffix="u"

for i in `seq 1 $3`;
do
  if [ $i -lt 10 ]; then #For padding a zero in name i.e. 01u
    target=$2$strDash$strZero$i$strSuffix
  else
    target=$2$strDash$i$strSuffix
  fi
  expect ./login.exp $target $1 $COUNT
  if [ $? -eq 1 ]; then
    COUNT=$((COUNT+1))
    if [ $COUNT -eq 11 ]; then
      ROUND=$((ROUND+1))
      COUNT=4
      if [ $ROUND -eq 4 ]; then
        echo "ALL DONE! Last PC to be assigned was: $target"
        exit 1
```

```
        fi
      fi
    fi
done

echo "Reached upper limit. Next values would be ROUND=$ROUND COUNT=$COUNT"
```

A.4 Makefile

```
SEARCH_DEPTH = 10000000
MAX_STEPS    = 10000
RANDOM_SEED  = 123
MEMORY_LIMIT = 32768 # In MB

verify:
	spin -a model.pml
	gcc -DMEMLIM=$(MEMORY_LIMIT) -O2 -DXUSAFE -DNFAIR=3 -DCOLLAPSE -w -o pan pan.c
	./pan -m$(SEARCH_DEPTH) -a -f
	rm pan*
	rm *.tmp

simulate:
	spin -p -s -r -X -v -n$(RANDOM_SEED) -l -g -u$(MAX_STEPS) model.pml

creator: creator.hs
	ghc $^
	rm creator.o
	rm creator.hi
```

Appendix B

Source code extracts of the library

B.1 inheritance_lock_api.h

```
/**
 * Inheritance Lock Protocol Header file
 * This is the API header use this, and only this, header file if you wish to use inheritance locks in your code.
 * Furthermore, you will need to link to the library during compilation.
 *
 * @author: W. David Frohlingsdorf
 * @date: 2016
 * @purpose: Dissertation, University of Glasgow
 * */

#ifndef INHERITANCE_LOCK_API
#define INHERITANCE_LOCK_API

/// * * * * Necessary includes * * * *
#include <pthread.h>

/// * * * * Types * * * *
typedef void *INHERIT_LOCK;


/// * * * * Prototypes * * * *

//Reserves enough memory for a lock, initaliases the lock and returns a pointer to it.
//If address of a pointer is given as an argument (otherwise NULL), this pointer will also be directed
//to the newly created lock. rec_lock allows a thread to acquire a lock more than once (nested)
//Returns NULL if an error occured
INHERIT_LOCK create_lock(INHERIT_LOCK*);
INHERIT_LOCK create_rec_lock(INHERIT_LOCK*);

//The calling thread tries to lock the given lock and might block until it is available.
//While the thread is blocked, previously acquired locks might be temporarly preempted according
//to the protocol's definition. Fails if thread tries to lock a non-recursive lock again.
//Returns 0 if successful, -1 otherwise
int acquire(INHERIT_LOCK);

//The calling thread release the given lock. Note that this also has to be the last lock that
//has been acquired by this thread as required by the protocol. Returns 0 if successful -1 otherwise
//(illegal release of lock). If not successful, the thread still owns the lock.
int release(INHERIT_LOCK);

//Frees the memory currently occupied by the lock object. The pointer to the lock, which address
//has to be given as an argument, will be set to NULL if successful. The function will fail if the
//lock is currently held by a thread or threads are waiting for it. Returns 0 if successful -1 otherwise.
int destroy_lock(INHERIT_LOCK*);

#endif /* INHERITANCE_LOCK_API */
```

B.2 acquire() Function

```
int acquire(INHERIT_LOCK l)
{
    LOCK *lock = (LOCK *)(l);
    THRD *t = getNode();
    THRD *root;
    int e;
    if(l == NULL)
    {
        if(VERBOSE)
            fprintf(stderr, "Can not acquire a NULL pointer!\n");
        if(LOG_LEVEL >= 1)
            logger("Can not acquire a NULL pointer!\n");
        return -1;
    }
    if(lock->recursive == 0 && lock->heldBy != NULL && stackContains(t, lock))
    {
        if(VERBOSE)
            fprintf(stderr, "Non-recursive lock has already been acquired before!\n");
        if(LOG_LEVEL)
            logger("Non-recursive lock has already been acquired before!\n");
        return -1;          //Non-recursive lock has already been acquired before
    }

    lockGraph();
    root = getRoot(lock);   //Check which thread is currently holding the lock
    if(root == NULL)        //No one is holding this lock
    {
        push(t, lock);
        lock->heldBy = t;
        if(LOG_LEVEL >= 3)
            logger("Acquired lock normally.\n");
        unlockGraph();
        return 0;
    }
    if(root == t)           //This thread is already holding the lock
    {
        push(t, lock);
        if(LOG_LEVEL >= 2)
            logger("Acquired lock inheritently.\n");
        unlockGraph();
        return 0;
    }
    t->waitsFor = lock;     //A different thread is holding the lock
    enqueue(lock, t);
    if(LOG_LEVEL >= 3)
        logger("Waiting for lock to become available.\n");
    e = pthread_cond_wait(&(t->alarm), &graphLock);
    if(e)
        abortWithExcept("An error occured in pthread_cond_wait(). Returned with", e);
    unlockGraph();
    return 0;
}
```

B.3 release() Function

```
//Releases a lock from the thread.
//Other than the API release, this internal release function requires the
//thread structure to be handed over in the second parameter. This is necessary
//as release_intern() might be called from the thread specific memory desctructor.
//@param The according lock
//@param The according thread
//@return -1 if an error occured, 0 otherwise
int release_intern(LOCK *lock, THRD *t)
{
    THRD *next;
    int e;
    if(peek(t) != lock)     //We can only release the lock from the top of the stack
    {
```

```
        if(VERBOSE)
            fprintf(stderr, "Tried to release a lock which is not on top of the stack!\n");
        if(LOG_LEVEL >= 1)
            logger("Tried to release a lock which is not on top of the stack!\n");
        return -1;
    }


    lockGraph();
    pop(t);
    if(t != lock->heldBy)    //Thread inherited this lock
    {
        if(LOG_LEVEL >= 2)
            logger("Released lock inheritently.\n");
        unlockGraph();
        return 0;
    }
    next = dequeue(lock);
    lock->heldBy = next;
    if(next!=NULL)           //If another thread is waiting for the lock, hand it over and wake up the thread
    {
        next->waitsFor = NULL;
        push(next, lock);
        e = pthread_cond_signal(&(next->alarm));
        if(e)
            abortWithExcept("An error occured while signaling condition variable!"
                            "pthread_cond_signal() returned with", e);
        if(LOG_LEVEL >= 3)
            logger("Transfered lock to next thread in queue.\n");
    }
    else if(LOG_LEVEL >= 3)
        logger("Released lock normally.\n");
    unlockGraph();
    return 0;
}
```

Appendix C

Scripts used in the tyche benchmark test

C.1 runO3.sh

```
#!/bin/bash

#Arguments:
#1 = applicationName
#2 = outputCsvFile
#3 = max_locks
#4 = max_threads_start
#5 = max_threads_interval
#6 = max_threads_iterations
#7 = rounds

current_date="`date +%Y-%m-%d-%H:%M:%S`";
echo $current_date

#Setup make a clean working environment
for i in 5 10 15 25 50;
do
  cp resultsO3/clean.csv resultsO3/$i.locks/tm.csv
  cp resultsO3/clean.csv resultsO3/$i.locks/inherit.csv
done

for (( j=0; j<100; j++));
do
  for i in 5 10 15 25 50;
  do
    ./runExperiment.sh testTM resultsO3/$i.locks/tm.csv $i 10 10 10 10
    ./runExperiment.sh testInherit resultsO3/$i.locks/inherit.csv $i 10 10 10 10
  done
done


#Post processing
for i in 5 10 15 25 50;
do
  python resultsO3/stat.py resultsO3/$i.locks/tm.csv resultsO3/$i.locks/inherit.csv "Max $i locks" "fig.bench.tych
done

current_date="`date +%Y-%m-%d-%H:%M:%S`";
echo $current_date
```

C.2 runExperiment.sh

```
#!/bin/bash

#Arguments:
#1 = applicationName
```

```
#2 = outputCsvFile
#3 = max_locks
#4 = max_threads_start
#5 = max_threads_interval
#6 = max_threads_iterations
#7 = rounds

upper=$(($4+$5*$6))
for ((j=0; j<$7; j++));
do
  for((i=$4; i<$upper; i=$((i+$5)) )); #i = Current max threads
  do
    ./$1 -t $i -l $3 > swap.tmp
    python outToCsv.py swap.tmp $2 $3 $i
  done
done

rm swap.tmp
```

C.3 outToCsv.py

```
import sys

#Arguments:
#resultFile outCsvFile locks threads

with open(sys.argv[1]) as f:
  line = f.readline()
  with open(sys.argv[2], "a") as o:
    o.write(line[36:].split("ns")[0] + "," + sys.argv[3] + "," + sys.argv[4] + "\n")
```

C.4 stat.py

```
import sys

#Arguments:
#1 firstCsvFile
#2 secondCsvFile
#3 Figure caption
#4 Figure label

#Hardcoded in file:
#  Size of test date (1000)
#  Whiskers (2%, 98%)

def h(a, b):
  return ((a + b)/2.0)

def printBegin(caption, label):
  print("\\begin{figure}")
  print("\\caption{" + caption + "}")
  print("\\label{" + label + "}")
  print("\\begin{tikzpicture}")
  print("  \\begin{axis}[")
  print("      boxplot/draw direction=y,")
  print("      xlabel={Max. thread. \\textcolor{blue}{TM}, \\textcolor{red}{Inheritance Locks}},")
  print("      ylabel={Time needed [ns]},")
  print("      xtick={4.5,10.5,16.5,22.5,28.5},")
  print("      xticklabels={20,40,60,80,100},")
  print("    ]")

def printEnd():
  print("  \\end{axis}")
  print("\\end{tikzpicture}")
  print("\\end{figure}")

def spacePlot():
  print("    \\addplot+[boxplot prepared={")
  print("        lower whisker=0, lower quartile=0,")
```

```
  print("        median=0, upper quartile=0,")
  print("        upper whisker=0}, white, solid]")
  print("        coordinates {};")

def readCsvFile(name, colour):
  listOut = []
  with open(name) as f:
    l = f.readline()

    # Hardcoded variables
    # length for each = 1000
    dic = {10:[], 20:[], 30:[], 40:[], 50:[], 60:[], 70:[], 80:[], 90:[], 100:[]}

    l = f.readline()
    while l != "":
      trip = l.strip("\n").split(",")
      v = int(trip[0])
      k = int(trip[2])
      dic[k].append(v)
      l = f.readline()

    for k in sorted(dic.iterkeys()):
      v = dic[k]
      v.sort()
      median = h(v[499],v[500])
      lower = h(v[249],v[250])
      upper = h(v[749],v[750])
      lowWhisk = h(v[19],v[20])
      highWhisk = h(v[979], v[980])
      s = "    \\addplot+[boxplot prepared={\n"
      s += "        lower whisker="+str(lowWhisk)+", lower quartile="+str(lower)+",\n"
      s += "        median="+str(median)+", upper quartile="+str(upper)+",\n"
      s += "        upper whisker="+str(highWhisk)+"}, "+colour+", solid]\n"
      s += "        coordinates {};"
      listOut.append(s)
  return listOut

#Main
blueData = readCsvFile(sys.argv[1], "blue")
redData = readCsvFile(sys.argv[2], "red")
first = True

printBegin(sys.argv[3], sys.argv[4])
for i in range(len(blueData)):
  if not first:
    spacePlot()
  else:
    first = False
  print(blueData[i])
  print(redData[i])
printEnd()
```

Appendix D

Scripts and source code used in the application stress test

D.1 pthread_interpose.c

```
/**
 * Dynamic shared library which when loaded
 * replaces all pthread mutecies calls to
 * inherit locks. Please check the readme
 * file in this directory for further details.
 *
 * Uses a generic map as defined in map.h
 *
 * @author: W. David Frohlingsdorf
 * @date: 2016
 * @purpose: Dissertation, University of Glasgow
 * */



/// * * * Thanks to * * *
/// -> stackoverflow.com/questions/3707358/get-all-the-thread-id-created-with-pthread-created-within-an-process

/// * * * Includes * * *

#define _GNU_SOURCE

#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <bits/pthreadtypes.h>
#include <dlfcn.h>

#include "map.h"
#include "inheritance_lock_api.h"

/// * * * Definitions and Macros * * *

#define VERBOSE 2      //0=Off, 1=On, 2=Only Warnings and Errors
#define USE_PTHREAD 0 //Use pthread mutex rather than inheritance locks
#define GLIBC_VERSION "GLIBC_2.3.2"
#define COND

/// * * * Shared variables * * *

extern pthread_mutex_t graphLock;

/// * * * Redirected functions * * *

#undef pthread_mutex_lock
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
{
  void *key = (void *)(mutex);
  INHERIT_LOCK lock = NULL;

  static int (*real_lock)(pthread_mutex_t *) = NULL;
  if(!real_lock)
    real_lock = dlsym(RTLD_NEXT, "pthread_mutex_lock");

  if(USE_PTHREAD || key == &graphLock)
    return real_lock(mutex);

  if(VERBOSE == 1)
    printf("Caught pthread_mutex_lock()\n");

  lock = map_get(key);
  if(lock == NULL)
  {
    lock = create_rec_lock(NULL);
    map_add(key, lock);
  }
  acquire(lock);
  return 0;
}

#undef pthread_mutex_unlock

int pthread_mutex_unlock(pthread_mutex_t *mutex)
{
  void *key = (void *)(mutex);
  INHERIT_LOCK lock = NULL;

  static int (*real_unlock)(pthread_mutex_t *) = NULL;
  if(!real_unlock)
    real_unlock = dlsym(RTLD_NEXT, "pthread_mutex_unlock");

  if(USE_PTHREAD || key == &graphLock)
    return real_unlock(mutex);

  if(VERBOSE == 1)
    printf("Caught pthread_mutex_unlock()\n");

  lock = map_get(key);
  release(lock);
  return 0;
}

#undef pthread_mutex_trylock

int pthread_mutex_trylock(pthread_mutex_t *mutex)
{
  static int (*real_trylock)(pthread_mutex_t *) = NULL;
  if(!real_trylock)
    real_trylock = dlsym(RTLD_NEXT, "pthread_mutex_trylock");

  if(USE_PTHREAD)
    return real_trylock(mutex);
  printf("TRYLOCK!\n");
  exit(0);
}

#undef pthread_mutex_destroy

int pthread_mutex_destroy(pthread_mutex_t *mutex)
{
  void *key = (void *)(mutex);
  INHERIT_LOCK lock = map_del(key);
  int r = 0;

  static int (*real_destroy)(pthread_mutex_t *) = NULL;
  if(!real_destroy)
    real_destroy = dlsym(RTLD_NEXT, "pthread_mutex_destroy");
```

```
  r = real_destroy(mutex);
  if(VERBOSE == 1)
    printf("Caught pthread_mutex_destroy()\n");
  if(lock != NULL)  //Only destroy if it has been initialised
    destroy_lock(&lock);
  return r;
}


/*********************CONDITION VARIABLE**********************/
#ifdef COND
/*************************************************************/

#undef pthread_cond_timedwait

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)
{
  void *key = (void *)(mutex);
  char *err;
  static int (*real_timedwait)(pthread_cond_t *, pthread_mutex_t *, const struct timespec *) = NULL;
  if(!real_timedwait)
    real_timedwait = dlvsym(RTLD_NEXT, "pthread_cond_timedwait", GLIBC_VERSION);
  if(VERBOSE && (!real_timedwait || (err = dlerror())))
    printf("Fatal error: dlvsym() could not be executed correctly!\n"
           "\tError message: %s\n"
           "\tPlease check that you provided the correct version!", err);


  if(VERBOSE && key != &graphLock)
  {
    struct timespec spec;
    struct timespec diff;
    clock_gettime(CLOCK_REALTIME, &spec);
    diff.tv_sec = abstime->tv_sec - spec.tv_sec;
    diff.tv_nsec = abstime->tv_nsec - spec.tv_nsec;
    printf("*****WARNING: pthread_cond_timedwait() has been called , timeout in: %ld[s],%ld[ns]*****\n",
           (long)(diff.tv_sec), diff.tv_nsec);
  }
  printf("timedwait\n");
  return real_timedwait(cond, mutex, abstime);
}

#undef pthread_cond_wait

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
{
  void *key = (void *)(mutex);
  char *err;
  static int (*real_wait)(pthread_cond_t *, pthread_mutex_t *) = NULL;
  if(!real_wait)
    real_wait = dlvsym(RTLD_NEXT, "pthread_cond_wait", GLIBC_VERSION);
  if(VERBOSE && (!real_wait || (err = dlerror())))
    printf("Fatal error: dlvsym() could not be executed correctly!\n"
           "\tError message: %s\n"
           "\tPlease check that you provided the correct version!", err);
  if(VERBOSE && key != &graphLock)
    printf("*****WARNING: pthread_cond_wait() has been called.*****\n");
  return real_wait(cond, mutex);
}

#undef pthread_cond_signal

int pthread_cond_signal(pthread_cond_t *cond)
{
  static int (*real_signal)(pthread_cond_t *) = NULL;
  char *err;
  if(!real_signal)
    real_signal = dlvsym(RTLD_NEXT, "pthread_cond_signal", GLIBC_VERSION);
  if(VERBOSE && (!real_signal || (err = dlerror())))
    printf("Fatal error: dlvsym() could not executed correctly!\n"
           "\tError message: %s\n"
           "\tPlease check that you provided the correct version!", err);
  return real_signal(cond);
```

```
}

#endif
```

D.2 run.sh

```
#!/bin/bash

# This script runs a predefined times
# gimp with the simple-unsharp-mask
# script and compares the output.

#Arguments:
#1 library

RUNS=500

if [ -f temp.txt ]; then
  rm temp.txt
fi

for i in `seq 1 $RUNS`;
do
  cp wiki_backup.png wiki.png
  export LD_PRELOAD=$1
  gimp -i -b '(simple-unsharp-mask "wiki.png" 5.0 0.5 0)' -b '(gimp-quit 0)'
  cmp -i 100 -l wiki.png wiki_ref.png | gawk '{printf "%08X %02X %02X\n", $1, strtonum(0$2), strtonum(0$3)}' | wc
done

echo ""
echo " * * * * * * * * * * "
echo ""
echo "From $RUNS"
grep '^0$' temp.txt | wc -l
echo "were successful."
echo ""
echo " * * * * * * * * * * "
echo ""

rm temp.txt
```

D.3 simple-unsharp-mask.scm

```
(define (simple-unsharp-mask filename
                             radius
                             amount
                             threshold)
  (let* ((image (car (gimp-file-load RUN-NONINTERACTIVE filename filename)))
         (drawable (car (gimp-image-get-active-layer image))))
    (plug-in-unsharp-mask RUN-NONINTERACTIVE image drawable radius amount threshold)
    (gimp-file-save RUN-NONINTERACTIVE image drawable filename filename)
    (gimp-image-delete image)))
```

Glossary

busy-wait A thread waits for the occurrence of an event by repeatedly checking if the event has occurred. 4, 44

collatz conjecture Mathematical conjecture which states that any positive integer n will eventually reach 1 if it is indefinitely applied to the rule: If even divide by 2, if odd times 3 plus 1. 32

conveyor resource Resource which lies on the path in the wait-for-graph between a process and an inherited resource. 17, 44

critical section Section of a program which has to be executed in atomic fashion with regard to one or more resources. 7, 24, 45

deadlock Situation in which two or more processes wait indefinitely for each other. 20, 27, 41, 45, 47

deadlock avoidance Deadlocks are avoided by keeping the system in a state that is guaranteed to be free of deadlocks, a so called safe state. 5

deadlock detection Deadlocks might occur but can be detected with a deadlock detection algorithm. This is usually done retrospectively, but preventive deadlock detection algorithms exist as well. 9

deadlock prevention The locking protocol makes the occurrence of deadlocks impossible. 5, 48

dining philosophers problem Famous illustration of the deadlock problem introduced by Dijkstra. 6, 40

forest Directed acyclic graph consisting of a number of components where each component is a tree. 15, 25

indegree Number of ingoing edges of a node in a directed graph. 14

linear temporal logic Modal temporal logic which allows to define conditions which are time concerned. For example: eventually x is true and y is always true. 19

livelock A deadlock, but the affected process remains in a busy state without doing any useful work. 5, 47

outdegree Number of outgoing edges of a node in a directed graph. 14

path Sequence of edges in a graph such that each subsequent edge starts at the node where the previous edge ended. 14

promela Verification modelling language which allows to define a concurrent process model. 19

root Source node in a tree, sink node in a transpose tree. 15, 25, 41

sink Node with no outgoing edges in a directed graph. 14, 25

source Node with no ingoing edges in a directed graph. 14

spin Model checker which allows to verify LTL properties in a Promela model. 19, 47

starvation A process has to wait for an unacceptable long time for a situation to occur and it is not guaranteed that the awaited situation ever occurs. 6, 11, 12, 20, 27, 34, 44, 45, 47

transaction Sequence of read/write operations on data which should be performed in a single atomic step, usually used in databases. 9

transactional memory Transactions in ordinary programming code (approach taken from databases), can be implemented as software or hardware. 9, 24, 27, 31, 47

transpose tree Directed acyclic graph where each node has an outdegree of one, apart from the root node. 15

tree Directed acyclic graph where each node has an indegree of one, apart from the root node. 15, 25

wait-for graph A directed graph, consisting of process and resources, showing which resources have been allocated and requested. 7, 13, 15, 20, 24, 31, 44, 48